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** (can be changed), and allow **duplicate values**.

They are written inside **square brackets [ ]**.

2. Creating a List

```
# 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

Indexing starts from **0** for the first element.

Negative indexing (-1) is used to access elements from the **end** of the list.

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

print(fruits[0])   # First element: apple

print(fruits[2])   # Third element: cherry

print(fruits[-1])  # Last element: date
```

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

1. Definition of Indexing

Indexing means accessing elements of a list using their **position (index)**.

In Python:

Positive indexing starts from **0** (first element).

Negative indexing starts from **-1** (last element).

2. Positive Indexing

Counts from **left to right**, starting at **0**.

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

print(fruits[0])  # First element: apple

print(fruits[2])  # Third element: cherry

print(fruits[4])  # Fifth element: mango
```

3. Negative Indexing

Counts from **right to left**, starting at **-1**.

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

print(fruits[-1])  # Last element: mango

print(fruits[-2])  # Second last element: date

print(fruits[-5])  # First element: apple
```

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

1. Definition of Slicing

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 the **beginning** or **end** of the list.

2. Examples of Slicing

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

print(numbers[1:4])   # Elements from index 1 to 3 → [20, 30, 40]

print(numbers[:3])    # First three elements → [10, 20, 30]

print(numbers[2:])    # Elements from index 2 to end → [30, 40, 50, 60, 70]

print(numbers[:])     # All elements → [10, 20, 30, 40, 50, 60, 70]
```

3. Negative Index Slicing

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

print(numbers[-3:])    # Last three elements → [50, 60, 70]

print(numbers[-6:-2])  # Elements from -6 to -3 → [20, 30, 40, 50]
```

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

Use the + operator to **combine two or more lists** into one.

```
list1 = [1, 2, 3]

list2 = [4, 5, 6]

result = list1 + list2

print(result)  # Output: [1, 2, 3, 4, 5, 6]
```

3. Repetition

Use the * operator to **repeat elements of a list multiple times**.

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

result = list1 * 3

print(result)  # Output: ['A', 'B', 'A', 'B', 'A', 'B']
```

4. Membership

Use **in** or **not in** to check if an element exists in a list.

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

print("apple" in fruits)     # True

print("mango" in fruits)     # False

print("grape" not in fruits) # True
```

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 the last element).

2. append()

Syntax: list.append(element)

Adds a **single element** at the **end** of the list.

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

fruits.append("cherry")

print(fruits)  # Output: ['apple', 'banana', 'cherry']
```

3. insert()

Syntax: list.insert(index, element)

Adds an element at the given **index**, shifting other elements to the **right**.

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

fruits.insert(1, "orange")

print(fruits)  # Output: ['apple', 'orange', 'banana', 'cherry']
```

4. remove()

Syntax: list.remove(element)

Removes the **first occurrence** of the specified element.

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

fruits.remove("banana")

print(fruits)  # Output: ['apple', 'cherry', 'banana']
```

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**.

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

fruits.pop(1)     # Removes element at index 1

print(fruits)     # Output: ['apple', 'cherry']

fruits.pop()      # Removes last element

print(fruits)     # Output: ['apple']
```

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.

Iteration 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)
```

Output:

```
apple
banana
cherry
date
```

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**.

You can use reverse=True to sort in **descending order**.

```
numbers = [5, 2, 9, 1, 7]

numbers.sort()  # Ascending

print("Ascending:", numbers)  # Output: [1, 2, 5, 7, 9]

numbers.sort(reverse=True)  # Descending

print("Descending:", numbers)  # Output: [9, 7, 5, 2, 1]
```

2. sorted() Function

The **sorted()** function returns a **new sorted list** without changing the original list.

Supports the **reverse=True** parameter.

```
numbers = [8, 3, 6, 1]

ascending = sorted(numbers)

descending = sorted(numbers, reverse=True)

print("Original:", numbers)     # Output: [8, 3, 6, 1]

print("Ascending:", ascending)  # Output: [1, 3, 6, 8]

print("Descending:", descending) # Output: [8, 6, 3, 1]
```

3. reverse() Method

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

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

fruits.reverse()
```

```
print("Reversed:", fruits)  # Output: ['cherry', 'banana', 'apple']
```

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

1. Addition of Elements

You can add elements to a list using:

append() → Adds **one item** at the end.

insert() → Adds an item at a **specific index**.

extend() → Adds **multiple items** from another list.

```
fruits = ["apple", "banana"]
fruits.append("cherry")              # Add at end
fruits.insert(1, "mango")            # Add at index 1
fruits.extend(["orange", "grape"])   # Add multiple elements
print(fruits)
# Output: ['apple', 'mango', 'banana', 'cherry', 'orange', 'grape']
```

2. Deletion of Elements

You can remove elements using:

remove(value) → Removes by **value**.

pop(index) → Removes by **position** and returns the removed value.

del → Deletes by **index** or the **entire list**.

clear() → Empties the list.

```
numbers = [10, 20, 30, 40, 50]
```

```
numbers.remove(30)       # Remove by value
numbers.pop(2)           # Remove element at index 2
del numbers[0]           # Delete element at index 0
print(numbers)           # Output: [20, 40, 50]
numbers.clear()          # Remove all elements
print(numbers)           # Output: []
```

3. Updating Elements

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

```
fruits = ["apple", "banana", "cherry"]
fruits[1] = "mango"  # Update index 1
print(fruits)        # Output: ['apple', 'mango', 'cherry']
```

4. Slicing a List

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

Negative indexing can also be used.

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

9. Introduction to tuples, immutability.

1. Introduction to Tuples

A **tuple** is an **ordered collection of elements**, similar to a list.

Unlike lists, tuples are **immutable**, meaning their elements **cannot be changed** once assigned.

Tuples are defined using **parentheses ( )** instead of square brackets [ ].

my_tuple = (10, 20, 30, 40)

print(my_tuple) # Output: (10, 20, 30, 40)

2. Immutability of Tuples

Tuples **cannot be changed** once created.

You **cannot** add, update, or delete individual elements.

t = (1, 2, 3)

t[1] = 10   # This will raise an error

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

t1 = (1, 2, 3)

t2 = (4, 5)

t3 = t1 + t2

print(t3) # Output: (1, 2, 3, 4, 5)

10. *Creating and accessing elements in a tuple.*

1. Creating Tuples

a) Tuple with Multiple Elements

t2 = (10, 20, 30, 40)

print(t2) # Output: (10, 20, 30, 40)

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)  # Output: (1, 'apple', 3.14, True)
```

c) Tuple without Parentheses (Tuple Packing)

You can create a tuple **without parentheses** — this is called **tuple packing**.

```
t4 = 5, 10, 15, 20

print(t4)  # Output: (5, 10, 15, 20)
```

2. Accessing Elements in a Tuple

You can access elements in a tuple using:

Indexing

Negative Indexing

Slicing

a) Indexing

```
t = (10, 20, 30, 40, 50)

print(t[0])  # First element → 10

print(t[3])  # Fourth element → 40
```

b) Negative Indexing

Negative indexing starts from the **end of the tuple**.

-1 → last element, -2 → second last, and so on.

```
t = (100, 200, 300, 400)
```

print(t[-1]) # Last element → 400

print(t[-3]) # Third last element → 200

c) Slicing

Slicing allows you to access a **range of elements** using: tuple[start:end]

Start index → inclusive

End index → exclusive

t = (10, 20, 30, 40, 50, 60)

print(t[1:4])   # Elements from index 1 to 3 → (20, 30, 40)

print(t[:3])   # First three elements → (10, 20, 30)

print(t[3:])   # Elements from index 3 to end → (40, 50, 60)

print(t[-3:])   # Last three elements → (40, 50, 60)

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** (original tuples remain unchanged).

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)

Output:

True

True

12. *Accessing tuple elements using positive and negative indexing.*

1. Positive Indexing

Positive indexing starts from **0** for the first element.

The index value **increases by 1** for each next element.

Example:

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

print(fruits[0])

print(fruits[1])

print(fruits[2])

print(fruits[4])

Output:

apple

banana

cherry

orange

2. Negative Indexing

Negative indexing starts from **-1** for the last element.

The index value **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])
```

Output:

```
orange

mango

cherry
```

13. *Slicing a tuple to access ranges of elements.*

1. Syntax of Tuple Slicing

Syntax:

```
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 Example – Accessing a Range of Elements

Example:

```
numbers = (10, 20, 30, 40, 50, 60, 70)

print(numbers[1:4])
```

Output:

```
(20, 30, 40)
```

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 (like 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

Syntax:

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

Example:

```
student = {
```

```
    "name": "John",
    "age": 21,
    "course": "Python"
}
print(student)
```

Output:

```
{'name': 'John', 'age': 21, 'course': 'Python'}
```

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)

Output:

{'name': 'Ravi', 'age': 21, 'course': 'BCA', 'grade': 'A'}

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)

Output:

{'name': 'Ravi', 'age': 22, 'course': 'BCA'}

b) Updating Multiple Values using update()

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

print(student)

Output:

{'name': 'Ravi', 'age': 23, 'course': 'BCA', 'grade': 'A+'}

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)
```

Output:

```
{'name': 'Ravi', 'course': 'BCA'}
```

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)
```

Output:

```
Removed: BCA

{'name': 'Ravi', 'age': 21}
```

c) Using popitem() Method

Removes and returns the **last inserted key–value pair**.

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

student.popitem()

print(student)
```

Output:

```
{'name': 'Ravi', 'age': 21}
```

d) Using clear() Method

Removes **all items** from the dictionary.

```
student.clear()

print(student)
```

Output:

```
{}
```

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())
```

Output:

```
dict_keys(['name', 'age', 'course'])
```

Use in Loop:

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

Output:

name

age

course

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())
```

Output:

dict_values(['Ravi', 21, 'BCA'])

Use in Loop:

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

Output:

Ravi

21

BCA

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())

Output:

dict_items([('name', 'Ravi'), ('age', 21), ('course', 'BCA')])

Use in Loop:

for key, value in student.items():

    print(key, ":", value)

Output:

name : Ravi

age : 21

course : BCA

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**.

You can also use the **keys()** method explicitly for clarity.

Example:

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

Output:

```
name
age
course
```

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)
```

Output:

```
Ravi
21
BCA
```

3. Iterating Over Key–Value Pairs

To access both **keys and values 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)
```

Output:

```
name : Ravi

age : 21

course : BCA
```

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)
```

Output:

{'name': 'Ravi', 'age': 21, 'course': 'BCA'}

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)

Output:

{'name': 'Ravi', 'age': 21, 'course': 'BCA'}

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**.

Steps:

Create an empty dictionary.

Loop through each character in the string.

If the character is already a key → increment its value by 1.
If not → add it to the dictionary with value 1.

Example:

```
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)
```

Output:

```
{'h': 1, 'e': 1, 'l': 3, 'o': 2, ' ': 1, 'w': 1, 'r': 1, 'd': 1}
```

20. *Defining functions in Python.*

A **function** in Python is a block of **reusable code** that performs a specific task. Functions make programs **modular, readable, and efficient**.

Syntax:

```
def function_name(parameters):

    # function body

    statement(s)
```

return value

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

Type 1: Without Parameters & Without Return Value

Definition:
A function that doesn't take input and doesn't return any value — it just performs a task.

Example:

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


# Function call
greet()
```

Output:

```
Hello, Welcome to Python Programming!
```

Type 2: With Parameters & Without Return Value

Definition:
Takes inputs (parameters) but doesn't return anything.

Example:

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

```
greet("Dharmesh")
```

Output:

```
Hello, Dharmesh Welcome to Python!
```

Type 3: Without Parameters & With Return Value

Definition:
Takes no input but returns a value.

Example:

```
def get_pi():
    return 3.14159

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

Output:

```
Value of Pi = 3.14159
```

Type 4: With Parameters & With Return Value

Definition:
Accepts parameters and returns a value.

Example:

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

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

```
print("Sum =", result)
```

Output:

```
Sum = 30
```

22. *Anonymous functions (lambda functions).*

1. Introduction

An **anonymous function** is a function **without a name**, created using the **lambda** keyword.
They are small, single-line functions that can take any number of arguments but only one expression.

Syntax:

```
lambda arguments: expression
```

Example:

```
square = lambda x: x * x

print(square(5))
```

Output:

```
25
```

23. *Introduction to Python modules and importing modules.*

1. What is a Module?

A **module** in Python is a file containing code such as **functions, variables, or classes**.
Modules help organize code, promote reusability, and keep programs manageable.

2. Importing Modules

Purpose:

Use built-in, user-defined, or external modules.

Make code reusable and organized.

Avoid rewriting the same code.

Ways to Import:

import module_name – Imports the whole module.

from module_name import function – Imports specific parts.

import module_name as alias – Gives the module a short name.

from module_name import * – Imports everything *(not recommended)*.

3. 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)

4. Benefits

Code reusability
Better organization
Easy maintenance
Access to powerful libraries

24. *Standard library modules: math, random.*

1. math Module

Provides **mathematical functions and constants**.

Function	Description
math.sqrt(x)	Returns the square root of x

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

Example:

```
import math
print(math.sqrt(25))
print(math.pi)
```

Output:

```
5.0
3.141592653589793
```

2. random **Module**

Used to **generate random numbers** or **select random elements**.

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

Function	Description
random.shuffle(list)	Randomly rearranges elements in a list
random.uniform(a, b)	Returns a random float between a and b

Example:

```
import random
print(random.randint(1, 10))
print(random.choice(["apple", "banana", "cherry"]))
```

25. *Creating custom modules.*

Steps to Create a Custom Module

Create a Python file with a **.py** extension (e.g., mymodule.py).

Write **functions, variables, or classes** inside it.

Save the file in the **same directory** as your main program.

Import and use it using the **import** statement.

Example:

mymodule.py

```
def greet(name):
    return f"Hello, {name}!"
```

main.py

```
import mymodule
```

```
print(mymodule.greet("Ravi"))
```

Output:

```
Hello, Ravi!
```