Module 7)
Python – Collections, functions and Modules(Theory)

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

A list in Python is like a container that can store many items such as numbers, words, or even other lists.

- Example: `my_list = [10, "apple", 3.5]`
- You can access items using their index (position number). For example:
  - `my_list[0]` → gives `10` (first item).
  - `my_list[1]` → gives `"apple"` (second item).

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

- **Positive Indexing:** Starts from the left side, beginning with `0`.
  - Example: `my_list[0]` → first element, `my_list[2]` → third element.
- **Negative Indexing:** Starts from the right side, beginning with `-1`.
  - Example: `my_list[-1]` → last element, `my_list[-2]` → second last element.

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

Slicing means taking out a part (range) of the list.

- Syntax: `list[start:end]`

  `start` → position to begin (included).

  `end` → position to stop (not included).

- Example:

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

- You can also skip items using a step:
  - `numbers[0:5:2]` → `[10, 30, 50]`

2. List Operations

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

- **Concatenation (+)**
  - Joining two or more lists together.

```
list1 = [1, 2, 3]
list2 = [4, 5]
result = list1 + list2
print(result)   # [1, 2, 3, 4, 5]
```

- **Repetition (*)**
  - Repeats the list elements multiple times.

```
fruits = ["apple", "banana"]
result = fruits * 2
print(result)   # ['apple', 'banana', 'apple', 'banana']
```

- **Membership (in / not in)**
  - Checks if an element is present in the list.

```
fruits = ["apple", "banana", "mango"]
print("apple" in fruits)     # True
print("orange" not in fruits)  # True
```

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

- **append()** → Adds an element at the **end** of the list.

```
fruits = ["apple", "banana"]
fruits.append("mango")
print(fruits)   # ['apple', 'banana', 'mango']
```

- **insert()** → Adds an element at a **specific position (index)**.

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

- **remove()** → Removes the **first matching value**.

```
fruits = ["apple", "banana", "apple"]
fruits.remove("apple")
print(fruits)   # ['banana', 'apple']   # only the first 'apple' is removed
```

- **pop()** → Removes an element by its **index** (default: last element).

```
fruits = ["apple", "banana", "mango"]
fruits.pop(1)
print(fruits)   # ['apple', 'mango']
```

3. Working with Lists

6.Iterating over a list using loops.

Iterating means going through each element one by one.

- Using **for loop**:

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

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

Output:

```
apple
banana
mango
```

- Using **while loop**:

```
fruits = ["apple", "banana", "mango"]
i = 0
while i < len(fruits):
    print(fruits[i])
    i += 1
```

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

1. **sort()** → Sorts the list in ascending order (changes the original list).

```
numbers = [5, 2, 9, 1]
numbers.sort()
print(numbers)   # [1, 2, 5, 9]
```

2. **sorted()** → Returns a new sorted list, original list stays same.

```
numbers = [5, 2, 9, 1]
new_list = sorted(numbers)
print(new_list)    # [1, 2, 5, 9]
print(numbers)     # [5, 2, 9, 1]
```

3. **reverse()** → Reverses the order of elements.

```
numbers = [5, 2, 9, 1]
numbers.reverse()
print(numbers)    # [1, 9, 2, 5]
```

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

1. **Addition (append/insert)**

```
fruits = ["apple", "banana"]
fruits.append("mango")          # add at end
fruits.insert(1, "orange")      # add at index 1
print(fruits)   # ['apple', 'orange', 'banana', 'mango']
```

2. **Deletion (remove/pop/del)**

```
fruits.remove("banana")   # remove by value
fruits.pop(0)             # remove by index
del fruits[1]             # delete element at index 1
print(fruits)
```

3. **Updating (change element)**

```
fruits = ["apple", "banana", "mango"]
fruits[1] = "orange"
print(fruits)   # ['apple', 'orange', 'mango']
```

4. **Slicing (get part of list)**

```
numbers = [10, 20, 30, 40, 50]
print(numbers[1:4])    # [20, 30, 40]
print(numbers[:3])     # [10, 20, 30]
print(numbers[::2])    # [10, 30, 50]
```

4. Tuple

9. Introduction to tuples, immutability.

- A **tuple** is a collection in Python, just like a list.
- The main difference is: **tuples are immutable** → once created, you cannot change, add, or remove items.
- Example:

  ```
  my_tuple = (1, 2, 3, "apple")
  ```

  Here, the tuple has numbers and a string.

10. Creating and accessing elements in a tuple.

- You can create a tuple using round brackets `()`.

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

- To access elements, use the **index** (starting from `0`):

  ```
  print(my_tuple[0])   # Output: 10
  print(my_tuple[2])   # Output: 30
  ```

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

- **Concatenation (joining tuples)**
  You can add two tuples together:

  ```
  t1 = (1, 2)
  t2 = (3, 4)
  result = t1 + t2
  ```

```
print(result)   # (1, 2, 3, 4)
```

- **Repetition**
  You can repeat a tuple using `*`:

```
t = (5, 6)
print(t * 3)   # (5, 6, 5, 6, 5, 6)
```

- **Membership (checking if an item exists)**
  Use `in` or `not in`:

```
my_tuple = (10, 20, 30)
print(20 in my_tuple)      # True
print(50 not in my_tuple)  # True
```

5. Accessing Tuples

12.Accessing tuple elements using positive and negative indexing.

- **Positive Indexing**:
  Index starts from **0** on the left.

```
my_tuple = (10, 20, 30, 40, 50)
print(my_tuple[0])   # 10 (first element)
print(my_tuple[3])   # 40 (fourth element)
```

- **Negative Indexing**:
  Index starts from **-1** on the right.

```
my_tuple = (10, 20, 30, 40, 50)
print(my_tuple[-1])   # 50 (last element)
print(my_tuple[-3])   # 30 (third element from end)
```

13.Slicing a tuple to access ranges of elements.

- You can get a **part of a tuple** using slicing:
  Syntax → `tuple[start:end]`
  (It takes elements from `start` index up to **end-1**).

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

- If you **skip start**, it begins from the first element:

```
print(my_tuple[:3])   # (10, 20, 30)
```

- If you **skip end**, it goes till the last element:

```
print(my_tuple[2:])   # (30, 40, 50, 60)
```

- You can also use **step value**:
  Syntax → `tuple[start:end:step]`

```
print(my_tuple[::2])   # (10, 30, 50) → every 2nd element
print(my_tuple[::-1])  # (60, 50, 40, 30, 20, 10) → reversed tuple
```

6. Dictionaries

14.Introduction to dictionaries: key-value pairs.

- A **dictionary** is a collection in Python used to store data in **key–value pairs**.
- Keys are unique, and each key has a value.
- Example:
- `student = {"name": "Alice", "age": 20, "grade": "A"}`
    - `"name"`, `"age"`, `"grade"` → **keys**
    - `"Alice"`, `20`, `"A"` → **values**

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

Accessing values:

```
print(student["name"])   # Alice
print(student.get("age"))  # 20
```

Adding a new key-value pair:

```
student["city"] = "Delhi"
print(student)  # {"name": "Alice", "age": 20, "grade": "A", "city": "Delhi"}
```

Updating a value:

```
student["age"] = 21
print(student)  # {"name": "Alice", "age": 21, "grade": "A", "city": "Delhi"}
```

Deleting an element:

```
del student["grade"]   # remove by key
print(student)  # {"name": "Alice", "age": 21, "city": "Delhi"}
```

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

- **keys()** → Returns all keys

  ```
  print(student.keys())   # dict_keys(['name', 'age', 'city'])
  ```

- **values()** → Returns all values

  ```
  print(student.values())  # dict_values(['Alice', 21, 'Delhi'])
  ```

- **items()** → Returns key-value pairs as tuples

  ```
  print(student.items())
  # dict_items([('name', 'Alice'), ('age', 21), ('city', 'Delhi')])
  ```

7. Working with Dictionaries

17.Iterating over a dictionary using loops.

You can loop through a dictionary in different ways:

Loop through keys:

```
student = {"name": "Alice", "age": 21, "city": "Delhi"}

for key in student:
    print(key)   # prints only keys
```

Loop through values:

```
for value in student.values():
    print(value)   # prints only values
```

Loop through both key and value:

```
for key, value in student.items():
    print(key, ":", value)
# Output:
# name : Alice
# age : 21
# city : Delhi
```

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

Using a loop:

```
keys = ["name", "age", "city"]
values = ["Alice", 21, "Delhi"]

my_dict = {}
for i in range(len(keys)):
    my_dict[keys[i]] = values[i]

print(my_dict)   # {'name': 'Alice', 'age': 21, 'city': 'Delhi'}
```

Using `zip()`:

```
keys = ["name", "age", "city"]
values = ["Alice", 21, "Delhi"]

my_dict = dict(zip(keys, values))
print(my_dict)   # {'name': 'Alice', 'age': 21, 'city': 'Delhi'}
```

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

We can use a dictionary to keep count of each character:

```
text = "banana"
count = {}

for char in text:
    if char in count:
        count[char] += 1
    else:
        count[char] = 1
```

```
print(count)   # {'b': 1, 'a': 3, 'n': 2}
```

8. Functions

20.Defining functions in Python.

- A **function** is a block of code that runs only when called.
- Functions make code reusable and organized.
- **Syntax:**

```
def function_name(parameters):
    # code
    return value
```

Example:

```
def greet():
    print("Hello, welcome to Python!")
greet()   # calling the function
```

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

- **Without parameters, without return value**

```
def say_hello():
    print("Hello!")
say_hello()
```

- **With parameters, without return value**

```
def greet(name):
    print("Hello", name)
greet("Alice")
```

- **Without parameters, with return value**

```
def give_number():
    return 10
print(give_number())   # 10
```

- **With parameters, with return value**

```
def add(a, b):
    return a + b
print(add(5, 3))   # 8
```

22.Anonymous functions (lambda functions).

- **Lambda** is a small, one-line function without a name.
- Syntax:
- `lambda arguments : expression`

Example:

```
square = lambda x: x * x
print(square(5))   # 25
```

Another example:

```
add = lambda a, b: a + b
print(add(3, 7))   # 10
```

9. Modules

23.Introduction to Python modules and importing modules.

- A **module** is a file containing Python code (functions, variables, classes).
- You use modules to reuse code.
- Importing a module:

```
import math
print(math.sqrt(16))   # 4.0
```

24.Standard library modules: math, random.

- **math module**

```
import math
print(math.pi)         # 3.141592653589793
print(math.factorial(5))  # 120
```

- **random module**

```
import random
print(random.randint(1, 10))  # random number between 1-10
print(random.choice(["red", "blue", "green"]))  # random choice
```

25.Creating custom modules.

- You can create your own module (a `.py` file) and use it in another program.

Example:
my_module.py

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

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

main.py

```
import my_module
```

```
print(my_module.add(5, 3))        # 8
print(my_module.greet("Alice"))  # Hello, Alice
```