❖ Module 7) Python – Collections, functions and Modules

➢ Accessing List:

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

Ans:

- A list is a collection of items in Python.
- List is orderd and mutable.
- You can create it using square brackets [].

1. Creating a list:

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

2. Accessing elements:

You can access items by their **index** (starting from 0).

```
print(fruits[0])  # Output: apple
print(fruits[2])  # Output: cherry
```

3. Accessing from the end:

Use negative index to start from the last item.

```
print(fruits[-1])  # Output: cherry
print(fruits[-2])  # Output: banana
```

4. Adding elements:

```
fruits.append("orange")   # Adds at the end
fruits.insert(1, "mango") # Adds at position 1
```

5. Changing elements:

```
fruits[0] = "grapes"  # Changes "apple" to "grapes"
```

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

Ans:

1. Positive Indexing

- Counts from the **start** of the list.
- First element = index 0, second = 1, and so on.

Example:

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

print(fruits[0])  # Output: apple
print(fruits[2])  # Output: cherry
```

2. Negative Indexing

- Counts from the **end** of the list.
- Last element = index -1, second last = -2, etc.

Example:

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

print(fruits[-1])  # Output: mango
print(fruits[-3])  # Output: banana
```

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

Ans:

1. Basic Slicing:

- **Syntax:**

  list[start:stop]
- **start** → index to **begin** (inclusive)
- **stop** → index to **end** (exclusive)

Example:

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

print(fruits[1:4])  # Output: ['banana', 'cherry', 'date']
```

Starts at index 1 (banana) and stops **before** index 4 (fig).

2. Slicing with Step:

- **Syntax:**

  list[start:stop:step]
- **step** → how many indices to skip

Example:

```
print(fruits[0:5:2])  # Output: ['apple', 'cherry', 'fig']
```

Picks every 2nd element from index 0 to 4.

3. Omitting Indices:

- Start omitted → starts from beginning
- Stop omitted → goes to the end

```
print(fruits[:3])   # ['apple', 'banana', 'cherry']
print(fruits[2:])   # ['cherry', 'date', 'fig']
print(fruits[:])    # ['apple', 'banana', 'cherry', 'date', 'fig'] (whole list)
```

➢ List Operations:

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

Ans:

1. Concatenation (Joining Lists):

- Use + to join two lists.

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

result = list1 + list2

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

2. Repetition (Repeating Elements):

- Use * to repeat a list multiple times.

```
numbers = [1, 2]

print(numbers * 3)  # Output: [1, 2, 1, 2, 1, 2]
```

3. Membership (Check if Item Exists):

- Use in to check, not in to check absence.

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

print("banana" in fruits)     # True

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

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

Ans:

1. append() – Add at the End:

- Adds **one item** at the end of the list.

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

fruits.append("cherry")

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

2. insert() – Add at Specific Position:

- Adds an item at a **specific index**.

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

fruits.insert(1, "mango")  # Add "mango" at index 1

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

3. remove() – Delete by Value:

- Removes the **first occurrence** of a value.

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

fruits.remove("banana")

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

4. pop() – Delete by Index:

- Removes item at a given **index** and returns it.
- If index is **not given**, removes the **last item**.

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

fruits.pop(1)  # Removes "banana"

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

➢ Working with Lists:

1.Iterating over a list using loops.

Ans:

1. Using for loop:

- Go through each element **one by one**.

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

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

2. Using for loop with index:

- Access elements **by index** using range().

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

for i in range(len(fruits)):
    print(f"Index {i} = {fruits[i]}")
ss
Index 2 = cherry
```

3. Using while loop:

- Iterate with a **counter**.

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

i = 0

while i < len(fruits):
    print(fruits[i])
    i += 1
```

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

Ans:

1. sort() – Sorts the list in-place:

- Changes the original list **permanently**.

```
numbers = [3, 1, 4, 2]

numbers.sort()       # Sort ascending
print(numbers)       # Output: [1, 2, 3, 4]

numbers.sort(reverse=True)  # Sort descending
print(numbers)              # Output: [4, 3, 2, 1]
```

2. sorted() – Returns a new sorted list:

- **Does not** change the original list.

```
numbers = [3, 1, 4, 2]

new_list = sorted(numbers)

print(new_list)      # Output: [1, 2, 3, 4]
print(numbers)       # Original list unchanged: [3, 1, 4, 2]

new_list_desc = sorted(numbers, reverse=True)
print(new_list_desc)  # Output: [4, 3, 2, 1]
```

3. reverse() – Reverse the list in-place:

- Reverses the **order of elements**.

```
numbers = [1, 2, 3, 4]

numbers.reverse()
print(numbers)       # Output: [4, 3, 2, 1]
```

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

Ans:

- List manipulation means performing operations on a list to change, access, or organize its elements in Python.

1. Addition:

- **Add at the end:** append()

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

fruits.append("cherry")  # ['apple', 'banana', 'cherry']
```

- **Add at specific position:** insert()

```
fruits.insert(1, "mango")  # ['apple', 'mango', 'banana', 'cherry']
```

- **Combine lists:** +

```
fruits + ["date", "fig"]  # ['apple', 'mango', 'banana', 'cherry', 'date', 'fig']
```

2. Deletion:

- **Remove by value:** remove()

```
fruits.remove("banana")  # ['apple', 'mango', 'cherry']
```

- **Remove by index:** pop()

```
fruits.pop(1)  # ['apple', 'cherry']
```

- **Delete slice:** del

```
del fruits[0:1]  # ['cherry']
```

3. Updating:

- **Change an element:**

```
fruits[0] = "grapes"  # ['grapes', 'cherry']
```

- **Change multiple elements using slicing:**

```
fruits[0:2] = ["apple", "banana"]  # ['apple', 'banana']
```

4. Slicing (Accessing a range):

- **Get a portion of the list:**

```
fruits = ["apple", "banana", "cherry", "date"]
print(fruits[1:3])  # ['banana', 'cherry']
```

- **With step:**

```
print(fruits[0:4:2])  # ['apple', 'cherry']
```

➢ Tuple:

1.Introduction to tuples, immutability.

Ans:

- A **tuple** in Python is a ordered, collection of items, similar to a list.
- It can store multiple values of different data types (integers, strings, floats, etc.).
- Tuples are written using **parentheses ( )**.

Example:

```
my_tuple = (10, "hello", 3.14, True)
print(my_tuple)
```

Immutability of Tuples:

- The key difference between a **list** and a **tuple** is that **tuples are immutable**.
- **Immutable** means **once created, the elements of a tuple cannot be changed, updated, or removed**.
- You can access elements in a tuple, but you **cannot modify** them.

Example:

```
my_tuple = (1, 2, 3)
print(my_tuple[0])   #Accessing works

my_tuple[0] = 10     #Error: 'tuple' object does not support item
```

Why use Tuples?

- They are **faster** than lists.
- Good for **storing fixed data** (like days of the week, coordinates, etc.).
- Tuples can be used as **keys in dictionaries** (since they are immutable).

2.Creating and accessing elements in a tuple.

Ans:

Creating a Tuple:

1. **Using parentheses ( )**

```
t1 = (10, 20, 30)
print(t1)   # (10, 20, 30)
```

2. **Without parentheses** (comma-separated values)

```
t2 = 1, 2, 3
print(t2)   # (1, 2, 3)
```

3. **Single-element tuple** → must include a comma

```
t3 = (5,)
print(t3)   # (5,)
```

4. **Empty tuple**

```
t4 = ()
print(t4)   # ()
```

Accessing Elements in a Tuple:

- **By index (0-based indexing)**

```
my_tuple = ("apple", "banana", "cherry")
print(my_tuple[0])   # apple
print(my_tuple[2])   # cherry
```

- **Negative indexing (from the end)**

```
print(my_tuple[-1])   # cherry
print(my_tuple[-2])   # banana
```

- **Slicing (access a range of elements)**

```
print(my_tuple[0:2])   # ('apple', 'banana')
print(my_tuple[:])     # ('apple', 'banana', 'cherry')
```

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

Ans:

- These operations are commonly used because tuples are **immutable.**
- you can't modify them, but you can create new ones using these operators.

1. Concatenation (joining tuples):

- You can join two or more tuples using the + operator.

```
t1 = (1, 2, 3)
t2 = (4, 5, 6)

result = t1 + t2
print(result)   # (1, 2, 3, 4, 5, 6)
```

2. Repetition (repeating tuples):

- Use the * operator to repeat a tuple multiple times.

```
t = ("Hi",) * 3

print(t)   # ('Hi', 'Hi', 'Hi')
```

3. Membership (checking if an element exists):

- Use in or not in operators.

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

print(20 in t)       # True
print(50 not in t)   # True
```

➢ Accessing Tuples:

1.Accessing tuple elements using positive and negative indexing.

Ans:

Accessing Tuple Elements:

1. Positive Indexing (from the start)

- Index starts at **0**.
- First element → index 0
- Last element → index n-1

2. Negative Indexing (from the end)

- Index starts at **-1** for the last element.
- Last element → index -1
- Second last → index -2

Example Tuple:

t = ("apple", "banana", "cherry", "date")

Index Positions:

- Positive Index → 0 1 2 3
- Tuple Elements → ("apple", "banana", "cherry", "date")
- Negative Index → -4 -3 -2 -1

Access Examples:

- print(t[0]) # apple (first element, positive index)
- print(t[2]) # cherry (third element, positive index)
- print(t[-1]) # date (last element, negative index)
- print(t[-3]) # banana (third from end, negative index)

2.Slicing a tuple to access ranges of elements.

Ans:

- Slicing means **extracting a range of elements** from a tuple using the format.
- : → extracts a slice.
- Start is **inclusive**, End is **exclusive**.
- Negative indexes and step values give more control.

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

- **start** → index where the slice begins (included).
- **end** → index where the slice ends (**excluded**).
- **step** → interval (default = 1).

Examples:

```
t = ("apple", "banana", "cherry", "date", "mango")
```

1. **Basic slicing (positive index):**

```
print(t[1:4])   # ('banana', 'cherry', 'date')
```

2. **Omitting start or end:**

```
print(t[:3])    # ('apple', 'banana', 'cherry')   # from start to index 2
print(t[2:])    # ('cherry', 'date', 'mango')     # from index 2 to end
```

3. **Negative indexes:**

```
print(t[-4:-1])   # ('banana', 'cherry', 'date')
```

4. **Using step:**

```
print(t[::2])    # ('apple', 'cherry', 'mango')   # every 2nd element
```

➢ Dictionaries:

1.Introduction to dictionaries: key-value pairs.

Ans:

- A **dictionary** is a collection of items in **key–value pairs**.
- Keys are **unique** and used to identify values.
- Values can be of **any data type** (string, int, list, tuple, another dictionary, etc.).
- Dictionaries are written inside **curly braces {}**.

Example:

```
student = {
   "name": "Hensi",
   "age": 22,
   "course": "Python"
}

print(student)
```

Output:

```
{'name': 'Hensi', 'age': 22, 'course': 'Python'}
```

Key–Value Pairs:

- **Key** → like a label (must be unique & immutable: string, number, tuple).
- **Value** → the data linked with the key (can be anything).

Example:

```
print(student["name"])   # Hensi
print(student["age"])    # 22
```

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

Ans:

1. Accessing Dictionary Elements

- You can access values using their **keys**.

```
student = {"name": "Hensi", "age": 22, "course": "Python"}

print(student["name"])      # Hensi
print(student.get("age"))   # 22
```

- .get() is safer because it won't throw an error if the key doesn't exist.

2. Adding Elements

- Just assign a value to a **new key**.

```
student["city"] = "Morbi"
print(student)# {'name': 'Hensi', 'age': 22, 'course': 'Python', 'city': 'Morbi'}
```

3. Updating Elements

- Reassign a value to an **existing key**.

```
student["age"] = 23
print(student)# {'name': 'Hensi', 'age': 23, 'course': 'Python', 'city': 'Morbi'}
```

4. Deleting Elements

- **del keyword** → removes a specific key-value pair.
- **.pop(key)** → removes and returns the value.
- **.clear()** → removes all items.

```
del student["course"]
print(student)   # {'name': 'Hensi', 'age': 23, 'city': 'Morbi'}

student.pop("city")
print(student)   # {'name': 'Hensi', 'age': 23}

student.clear()
print(student)   # {}
```

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

Ans:

1. keys():

- Returns all the **keys** of the dictionary.
- Output is a special dict_keys object (like a list of keys).

```
student = {"name": "Hensi", "age": 22, "course": "Python"}

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

# Loop through keys:

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

2. values():

- Returns all the **values** of the dictionary.
- Output is a dict_values object.

```
print(student.values())   # dict_values(['Hensi', 22, 'Python'])

# Loop through values:

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

3. items():

- Returns all the **key–value pairs** as tuples inside a list-like object.

```
print(student.items())

# dict_items([('name', 'Hensi'), ('age', 22), ('course', 'Python')])

# Loop through key–value pairs:

for k, v in student.items():
    print(k, ":", v)
```

➢ Working with Dictionaries:

1.Iterating over a dictionary using loops.

Ans:

- for key in dict: → iterates over **keys**.
- for key in dict.keys(): → explicitly keys.
- for value in dict.values(): → values.
- for key, value in dict.items(): → key-value pairs.

Example:

```
student = {"name": "Hensi", "age": 22, "course": "Python"}
```

1. Iterating over Keys:

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

Output:

```
name
age
course
```

2. Iterating over Values:

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

Output:

```
Hensi
22
Python
```

3. Iterating over Key–Value Pairs:

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

Output:

```
name : Hensi
age : 22
course : Python
```

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

Ans:

- **Loop method**:
  manual, flexible.

- **zip() method**:
  cleaner and Pythonic.

Example Lists:

```
keys = ["name", "age", "course"]
values = ["Hensi", 22, "Python"]
```

1. Using a Loop:

```
my_dict = {}

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

print(my_dict)
```

Output:

```
{'name': 'Hensi', 'age': 22, 'course': 'Python'}
```

2. Using zip():

- zip() pairs elements from two lists together.

```
my_dict = dict(zip(keys, values))
print(my_dict)
```

Output:

```
{'name': 'Hensi', 'age': 22, 'course': 'Python'}
```

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

Ans:

- Use a dictionary to **store characters as keys** and their **counts as values**.
- Loop through the string and **increment the count** for each character.
- .get() makes it shorter and cleaner.

Example String:

text = "hello world"

Step 1: Using a Loop:

```
char_count = {}

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}

Step 2: Using dict.get() (Cleaner):

```
char_count = {}

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

print(char_count)
```

- .get(char, 0) → returns **0** if the key doesn't exist, otherwise returns the current value.

➢ Functions:

1.Defining functions in Python.

Ans:

- A **function** in Python is a reusable block of code that performs a specific task.
- You can define a function using the def keyword.

Basic Syntax:

```
def function_name(parameters):
        # function body
        # some code
        return value
```

- **Def**:

  keyword to define a function

- **function_name**:

  name of your function (should be meaningful)

- **parameters**:

  values passed into the function (optional)

- **return**:

  gives back a value (optional)

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

Ans:

- There are four types of function:

1.Function Without Parameters and Without Return Value:

- This type of function does **not take any input** and **does not return anything**.
- It only performs some task (e.g., printing a message).

Example:

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

# Function call
greet()
```

Output:

Hello, welcome to Python!

2.Function With Parameters and Without Return Value:

- This function **accepts input (parameters)** but **does not return** any value.
- It usually performs operations and displays the result.

Example:

```
def add(a, b):
    result = a + b
    print("Sum:", result)

# Function call with arguments
add(5, 10)
```

Output:

Sum: 15

3.Function Without Parameters and With Return Value:

- This type of function **does not take any input**, but **returns a value** to the caller.

Example:

```
def give_number():
    return 100

# Function call and storing the return value
num = give_number()
print("Returned number:", num)
```

Output:

Returned number: 100

4.Function With Parameters and With Return Value:

- This is the most commonly used type.
- The function **accepts parameters** and **returns a value** after processing.

Example:

```
def multiply(x, y):
    return x * y

# Function call and storing the returned value
product = multiply(4, 6)
print("Product:", product)
```

Output:

Product: 24

3. Anonymous functions (lambda functions).

Ans:

- An **anonymous function** is a function **without a name**.
- In Python, we create anonymous functions using the **lambda** keyword,so they're also called **lambda functions**.

Basic Syntax:

lambda arguments: expression

- **Lambda**:

  keyword to define the anonymous function

- **Arguments**:

  input values (like parameters)

- **Expression**:

  single line that returns a value (no return keyword needed)

Example : Lambda Function for Addition

```
add = lambda a, b: a + b

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

➢ Modules:

1.Introduction to Python modules and importing modules.

Ans:

What is a Module?

- A **module** is a **file containing Python code** (functions, classes, or variables) that you can **reuse** in other programs.
- It helps **organize code** and **avoid repetition**.

Types of Modules:

1. **Built-in Modules**:

   Already provided by Python (e.g., math, random, datetime).

2. **User-defined Modules**:

   Created by you.

3. **External Modules**:

   Installed using tools like pip (e.g., pandas, requests).

➢ Importing Modules:

1.Import the entire module

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

2.Import with an alias

```
import math as m
print(m.pi)
```

3.Import specific functions

```
from math import sqrt, pi
print(sqrt(25))
print(pi)
```

4.Import all (not recommended)

```
from math import *
print(sin(0))
```

- **User-Defined Module Example:**

mymodule.py

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

main.py

```
import mymodule
print(mymodule.greet("Hensi"))
```

Output:

```
Hello, Hensi!
```

2. Standard library modules: math, random.

Ans:

- **math Module:**

- Used for **mathematical operations**.

use:

import math

☑**Common Functions**:

- math.sqrt(x) → Square root
- math.pow(x, y) → x raised to the power y
- math.floor(x) → Rounds down to nearest integer
- math.ceil(x) → Rounds up to nearest integer
- math.pi → Value of pi (3.14159...)
- math.factorial(x) → Factorial of x

- **random Module:**

- Used to **generate random numbers**.

use:

import random

☑**Common Functions**:

- random.randint(a, b) → Random integer between a and b
- random.random() → Random float between 0.0 and 1.0
- random.choice(seq) → Random element from a list/tuple/string
- random.shuffle(seq) → Shuffles a list in place

3. Creating custom modules.

Ans:

- A **module** is a file that contains **Python code** (functions, variables, classes) that you can **import** and reuse in other programs.

- Every .py file is a module.

➢ **Steps to Create a Custom Module**

1.Create a Python file (module):

- **mymodule.py**

```
# mymodule.py

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

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

2.Import and Use the Module in Another File:

- **main.py**

```
import mymodule

print(mymodule.greet("Hensi"))    # Hello, Hensi!
print(mymodule.add(5, 3))         # 8
```

- **Other Import Methods**

- **Import specific functions**

```
from mymodule import greet
print(greet("User"))
```

- **Import with alias**

```
import mymodule as mm
print(mm.add(2, 4))
```