MODULE 14) PYTHON-COLLECTIONS,FUNCTIONS AND MODULES IN PYTHON

1.Accessing List

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

Ans: A list in Python is an ordered collection of elements that can store different data types such as integers, strings, or floats.
Lists are created using square brackets [ ], and items are separated by commas.
Elements in a list can be accessed using their index positions — the index number inside square brackets after the list name.

Example:

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

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

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

- **Ans:** Indexing is used to access individual elements from a list.
    - **Positive Indexing:** Starts from **0** for the first element and increases sequentially.
      Example: fruits[1] → "banana"
    - **Negative Indexing:** Starts from **-1** for the last element and moves backward.
      Example: fruits[-1] → "cherry"
      This allows easy access to elements from both the beginning and end of the list.

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

Ans: Slicing is a technique used to access a subset of elements from a list.
It uses the syntax list[start:end], where the start index is included, and the end index is excluded.
You can also skip indices to include defaults:

- list[:end] → from start to end index
- list[start:] → from start index to end of list
- list[start:end:step] → with custom step value

Example:

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

print(numbers[1:4])   # [20, 30, 40]

print(numbers[-3:])   # [40, 50, 60]
```

2. List Operations

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

Ans: Python lists support several basic operations that make them flexible and easy to use.

Concatenation (+):
Used to join two or more lists together to form a new list.

- list1 = [1, 2, 3]
- list2 = [4, 5, 6]
- result = list1 + list2
- print(result) # [1, 2, 3, 4, 5, 6]

Repetition (*):
Used to repeat the elements of a list multiple times.

- nums = [10, 20]
- print(nums * 3) # [10, 20, 10, 20, 10, 20]

Membership (in, not in):
Used to check whether an element exists in a list.

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

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

Ans: Python provides several built-in methods to modify and manage list elements.

append(item) → Adds an element to the end of the list.

- fruits = ["apple", "banana"]
- fruits.append("cherry")
- print(fruits) # ['apple', 'banana', 'cherry']

insert(index, item) → Inserts an element at a specific position.

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

remove(item) → Removes the first occurrence of the specified element.

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

pop(index) → Removes and returns an element at the given index. If no index is given, it removes the last element.

- fruits.pop()

print(fruits) # ['apple', 'mango']

3. Working with Lists

1. Iterating over a list using loops.

Ans: Iteration means accessing each element of a list one by one using loops.
The most common way to iterate through a list is by using a for loop or a while loop.

Example:

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

# Using a for loop

for fruit in fruits:

  print(fruit)

# Using a while loop

i = 0

while i < len(fruits):

  print(fruits[i])

  i += 1
```

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

Ans: Python provides built-in methods to arrange and reverse the order of list elements.

- sort() → Sorts the list in ascending order permanently.
- sorted() → Returns a new sorted list without changing the original.
- reverse() → Reverses the order of the list elements.

Example:

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

numbers.sort()

print("Sorted list:", numbers)          # [10, 20, 30, 40, 50]

reversed_list = sorted(numbers, reverse=True)

print("Sorted in descending order:", reversed_list)  # [50, 40, 30, 20, 10]

numbers.reverse()

print("Reversed list:", numbers)        # [50, 40, 30, 20, 10]
```

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

Ans: List manipulation means changing the contents of a list by adding, removing, updating, or extracting parts of it.

Addition:
Add elements using append() (end), insert() (specific position), or extend() (combine lists).

- my_list = [1, 2, 3]
- my_list.append(4) # [1, 2, 3, 4]
- my_list.insert(1, 10) # [1, 10, 2, 3, 4]

Deletion:
Remove elements using remove(), pop(), or del.

- my_list.remove(10) # Removes first occurrence of 10
- my_list.pop(2) # Removes element at index 2

Updating:
Change a value directly by assigning a new value to an index.

- my_list[0] = 100 # [100, 2, 4]

Slicing:
Access a range of elements using slicing list[start:end].

```
print(my_list[1:3])  # Prints elements between index 1 and 2
```

4. Tuple Theory:

1. Introduction to tuples, immutability.

Ans: A tuple in Python is an ordered collection of elements, similar to a list, but immutable, meaning its elements cannot be changed, added, or removed after creation.
Tuples are defined using parentheses ( ), and elements are separated by commas.
They are often used to store fixed collections of data that should not be modified.

Example:

my_tuple = (10, 20, 30, "apple")

Because tuples are immutable, operations that try to modify their contents (like append() or assignment) are not allowed.
This immutability makes tuples faster and more secure for storing constant data.

2. Creating and accessing elements in a tuple.

Ans: Tuples can be created by enclosing elements within parentheses ( ).
Elements can be accessed using indexing, just like lists.
Indexing starts from 0 for the first element and **-1** for the last element.

Example:

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

print(fruits[0])    # Access first element → apple

print(fruits[-1])   # Access last element → mango

print(fruits[1:3])  # Slicing → ('banana', 'cherry')
```

This allows easy access to any element or range of elements in a tuple.

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

Ans: Tuples support several basic operations similar to lists, but without modification:

Concatenation (+): Combines two tuples into one.

- tuple1 = (1, 2, 3)
- tuple2 = (4, 5)
- result = tuple1 + tuple2

- print(result) # (1, 2, 3, 4, 5)

Repetition (*): Repeats the elements of a tuple multiple times.

- nums = (10, 20)
- print(nums * 2) # (10, 20, 10, 20)

Membership (in, not in): Checks if an element exists in a tuple.

- fruits = ("apple", "banana", "cherry")
- print("apple" in fruits) # True

print("mango" not in fruits) # True

5. Accessing Tuples

1. Accessing tuple elements using positive and negative indexing

Tuples are ordered collections, so every element has a specific position known as an index.
You can access elements using positive and negative indexing:

- Positive Indexing:
  Starts from 0 for the first element and increases by 1 for each next element.
  Example:
- colors = ("red", "green", "blue", "yellow")
- print(colors[0]) # red
- print(colors[2]) # blue
- Negative Indexing:
  Starts from -1 for the last element and moves backward.
  Example:
- print(colors[-1]) # yellow
- print(colors[-3]) # green

This makes it easy to access elements from both ends of a tuple.

2. Slicing a tuple to access ranges of elements

Slicing is used to access a portion (range) of a tuple instead of single elements.
The syntax is:

tuple_name[start:end]

- start → index where the slice begins (inclusive).
- end → index where the slice stops (exclusive).

Example:

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

print(fruits[1:4]) # ('banana', 'cherry', 'mango')

print(fruits[:3]) # ('apple', 'banana', 'cherry')

print(fruits[-3:]) # ('cherry', 'mango', 'orange')

6. Dictionaries

1. Introduction to dictionaries: key–value pairs

A **dictionary** in Python is an unordered collection of data stored in key–value pairs. Each key in a dictionary is unique and is used to access its corresponding value. Dictionaries are written using curly braces {}, with pairs separated by commas.

Example:

student = {"name": "Aayushi", "age": 20, "course": "Python"}

Here,

- "name", "age", "course" → keys
- "Aayushi", 20, "Python" → values

Dictionaries are useful for representing structured data where each piece of information is identified by a unique key.

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

- **Accessing elements:**
  Use the key name inside square brackets or the .get() method.
- print(student["name"]) # Aayushi
- print(student.get("age")) # 20
- **Adding elements:**
  You can add new key–value pairs by assigning a new key.
- student["city"] = "Delhi"
- **Updating elements:**
  You can update the value of an existing key.
- student["age"] = 21
- **Deleting elements:**
  Use del or .pop() to remove an item.
- del student["course"]
- student.pop("city")

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

Dictionaries come with built-in methods to easily access data:

- **keys()** → Returns all the keys in the dictionary.

- print(student.keys()) # dict_keys(['name', 'age'])
- **values()** → Returns all the values.
- print(student.values()) # dict_values(['Aayushi', 21])
- **items()** → Returns all key–value pairs as tuples.
- print(student.items())
- # dict_items([('name', 'Aayushi'), ('age', 21)])

7. Working with Dictionaries

1. Iterating over a dictionary using loops

You can use **loops** (especially the for loop) to go through all the elements of a dictionary.
By default, looping over a dictionary iterates through its **keys**, but you can also access **values** or **key–value pairs** using methods like .values() and .items().

Example:

```
student = {"name": "Aayushi", "age": 20, "course": "Python"}

# Iterating through keys

for key in student:

  print(key, ":", student[key])

# Iterating through key-value pairs

for key, value in student.items():

  print(key, "→", value)
```

This allows you to easily **access and display** all information stored in a dictionary.

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

Dictionaries can be created by combining two separate lists — one containing **keys** and the other containing **values**.

1. **Using a for loop:**

   ```
   keys = ["name", "age", "course"]

   values = ["Aayushi", 20, "Python"]

   my_dict = {}

   for i in range(len(keys)):

     my_dict[keys[i]] = values[i]

   print(my_dict)
   ```

2. **Using the zip() function:**

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

   print(my_dict)
   ```

3. Counting occurrences of characters in a string using dictionaries

A dictionary can be used to count how many times each character appears in a string. Each character becomes a key, and its count becomes the value.

Example:

```
text = "banana"

count = {}

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

Output:

```
{'b': 1, 'a': 3, 'n': 2}
```

8. Functions

1. Defining functions in Python

A function in Python is a block of reusable code designed to perform a specific task. Functions make programs modular, organized, and easy to maintain.

They are defined using the def keyword followed by the function name and parentheses ().

Syntax:

```
def function_name():
  # block of code
```

Example:

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

You can call a function by using its name followed by parentheses:

```
greet()
```

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

Functions can be classified based on whether they take parameters and/or return values:

Without parameters and without return value:
The simplest type of function that just performs an action.

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

With parameters and without return value:
Takes input but doesn't return anything.

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

Without parameters but with return value:
Doesn't take any input but returns a value.

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

With parameters and with return value:
Takes input and returns a result — most flexible form.

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

3. Anonymous functions (lambda functions)

A lambda function is a small, anonymous (unnamed) function in Python.
It is defined using the keyword lambda and is usually used for short, simple operations.

Syntax:

lambda arguments: expression

Example:

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

You can also create lambda functions with **multiple parameters**:

```
add = lambda a, b: a + b
print(add(3, 4))   # Output: 7
```

9. Modules

1. Introduction to Python modules and importing modules

A module in Python is simply a file containing Python code — functions, classes, or variables — that can be reused in other programs.
Modules help keep code organized, reusable, and easy to maintain.

To use a module, you import it using the import statement.

Example:

```
import math

print(math.sqrt(25))   # Output: 5.0
```

You can also import specific parts of a module:

```
from math import sqrt

print(sqrt(16))       # Output: 4.0
```

Or give a module a shorter name (alias):

```
import math as m

print(m.pi)
```

2. Standard library modules: math, random

Python provides a large set of built-in modules known as the Standard Library, which includes modules like math and random.

math module:
Used for performing mathematical operations.

```
import math

print(math.sqrt(9))     # 3.0

print(math.factorial(5))# 120

print(math.pi)        # 3.141592653589793
```

random module:
Used for generating random numbers and making random selections.

```
import random

print(random.randint(1, 100))  # Random integer between 1 and 100
```

```
print(random.choice(["apple", "banana", "cherry"]))  # Random item from list
```

3. Creating custom modules

You can create your own module by writing Python code in a separate file (with a .py extension) and importing it into another program.

Example:

custom_module.py

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

main_program.py

```
import custom_module
print(custom_module.greet("Aayushi"))
```

Output:

Hello, Aayushi!