Module : Python – Collections, functions and Modules

❖ Accessing List

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

➢ What is a List?

- A list is a built-in data structure in Python.
- It is used to store multiple items in a single variable.
- Lists are ordered, changeable (mutable), and allow duplicate values.

How to Create a List

You create a list by placing items inside square brackets [], separated by commas:

Example:

```
my_list = [10, 20, 30, 40]
```

A list can contain any data type: integers, strings, booleans, even other lists.

You can also create an empty list: empty_list = []

How to Access List Elements

1. Indexing

- Each item in a list has an index number.
- Indexing starts from 0:
  - First element → index 0
  - Second element → index 1, and so on.

Example:

```
my_list[0]          # gives the first item
```

2. Negative Indexing

- Negative indexes start from the end of the list:
  - Last element → index -1
  - Second last → -2, and so on.

Example:

```
my_list[-1]                    # gives the last item
```

3. Slicing

- Slicing is used to access a range of elements:

```
syntax: my_list[start : end]
```

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

➢ What is Indexing

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

- Indexing starts from 0.
- You can use positive or negative numbers to refer to elements.

Positive Indexing

- Begins from the left (start of the list).
- The first element has index 0.
- The second element has index 1, and so on.

Example:

```
fruits = ["apple", "banana", "cherry", "date"]
Index:      0          1          2        3
```

Code:

```
fruits[0] → "apple"
fruits[2] → "cherry"
```

Negative Indexing

- Begins from the right (end of the list).
- The last element has index -1.
- The second last is -2, and so on.

Example:

```
fruits = ["apple", "banana", "cherry", "date"]
Index:      -4          -3          -2          -1
```

Code:

```
fruits[-1] → "date"
fruits[-3] → "banana"
```

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

➢ Slicing means extracting a subset (or part) of a list using a range of indexes.

Python slicing uses the colon (:) operator.

```
list [start : end]
```

start: index to begin slicing (inclusive)

end: index to stop slicing (exclusive)

The element at the end index is not included.

If start is omitted → defaults to 0

If end is omitted → goes till end of list

❖ List Operations

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

1. Concatenation (+):
   - Concatenation means joining two or more lists together.
   - The + operator is used for concatenation.
   - It does not modify the original lists but creates a new list.

   Example: [1, 2] + [3, 4] → [1, 2, 3, 4]

2.Repetition (*):

- Repetition means repeating the elements of a list multiple times.
- The * operator is used for repetition.
- It creates a new list with repeated elements.

  Example: [1, 2] * 3 → [1, 2, 1, 2, 1, 2]

3.Membership (in, not in)

- Membership operators are used to check whether an element exists in a list or not.
- in returns True if the element is present, otherwise False.
- not in returns True if the element is absent.

  Example:

- 3 in [1, 2, 3] → True
- 5 not in [1, 2, 3] → True

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

➢ 1. append()

- Used to add a single element at the end of the list.
- Syntax: list.append(element)
- It modifies the original list.
- Example:
    - numbers = [1, 2, 3]
    - numbers.append(4) → [1, 2, 3, 4]

2. insert()

- Used to insert an element at a specific index (position) in the list.
- Syntax: list.insert(index, element)
- It shifts the existing elements to the right.
- Example:
    - numbers = [1, 2, 4]
    - numbers.insert(2, 3) → [1, 2, 3, 4]

3. remove()

- Used to remove the first occurrence of a specific element from the list.
- Syntax: list.remove(element)
- If the element does not exist, it raises a ValueError.
- Example:
    - numbers = [1, 2, 3, 2]
    - numbers.remove(2) → [1, 3, 2]

4. pop()

- Used to remove and return an element from the list.
- By default, it removes the last element if no index is given.
- Syntax: list.pop([index])
- Raises IndexError if the index is out of range.
- Example:
  - numbers = [1, 2, 3]
  - numbers.pop() → removes 3, list becomes [1, 2]
  - numbers.pop(0) → removes 1, list becomes [2]

❖ Working with Lists

1.Iterating over a list using loops.

1. Using for loop

- The for loop is the most common way to iterate through a list.
- It directly retrieves each element of the list one by one.
- Example (conceptual):
  - for element in list: → accesses every element in the list.

2. Using for loop with range() and indexing

- Instead of directly iterating, we can use indices with range() to access elements.

- This gives more control (useful when you need both index and value).
- Example (conceptual):
    - for i in range(len(list)): → access elements as list[i].

3. Using while loop

- A while loop can also be used for iteration by manually controlling the index.
- Example (conceptual):
    - Initialize an index → i = 0
    - Loop while i < len(list)
    - Access element as list[i], then increase i.

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

1. sort() method:

- Used to sort the elements of a list in ascending order by default.
- Syntax: list.sort()
- It modifies the original list (in-place sorting).
- You can also pass arguments:
    - reverse=True → sorts in descending order.

    - key=function → sorts using a custom rule.

Example (conceptual):

- [3, 1, 2].sort() → [1, 2, 3]

2. sorted() function:

- A built-in function that returns a new sorted list from the given iterable.
- Syntax: sorted(iterable, key=None, reverse=False)
- It does not modify the original list.

Example (conceptual):

- sorted([3, 1, 2]) → [1, 2, 3]
- Original list remains unchanged.

3. reverse() method:

- Used to reverse the order of elements in a list.
- Syntax: list.reverse()
- It modifies the original list (in-place reversal).
- Note: This is different from sorting in descending order—it simply flips the current order.

Example (conceptual):

- [1, 2, 3].reverse() → [3, 2, 1]

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

1. Addition (Inserting Elements into a List)

- Lists in Python are dynamic, meaning elements can be added at any time.
- Methods used for addition:
    - append() → adds an element at the end.
    - insert(index, element) → adds an element at a specific position.
    - extend(iterable) → adds multiple elements from another list or iterable.

2. Deletion (Removing Elements from a List)

- Elements can be deleted in different ways:
    - remove(element) → removes the first occurrence of the specified element.
    - pop([index]) → removes and returns the element at the given index (default last element).
    - del list[index] → deletes element(s) at a specific index or a slice of elements.
    - clear() → removes all elements, leaving an empty list.

3. Updating (Changing Elements in a List)

- Since lists are mutable, their elements can be modified directly.
- Access an element by its index and assign a new value.
- Example (conceptual):
    - list[2] = 50 → updates the element at index 2.

4. Slicing (Extracting Subsets of a List)

- Slicing allows accessing a portion (sub-list) of the list.
- Syntax: list[start:end:step]
    - start → starting index (default: 0).
    - end → stopping index (excluded).
    - step → interval between elements (default: 1).
- Example (conceptual):
    - list[1:4] → extracts elements from index 1 to 3.
    - list[::-1] → reverses the list.

❖ Tuple

1.Introduction to tuples, immutability.

1. Introduction to Tuples

- A tuple in Python is an ordered collection of elements, similar to a list.
- Tuples can store different types of data (integers, strings, floats, etc.).
- They are written inside parentheses ( ) separated by commas.
    - Example: numbers = (1, 2, 3)
- Tuples allow indexing, slicing, iteration, and nested structures (tuples inside tuples).

2. Immutability of Tuples

- The most important property of tuples is immutability.
- Immutability means:
    - Once a tuple is created, its elements cannot be changed, added, or removed.
    - You cannot update an element at a specific index.
- For example:
    - t = (10, 20, 30)
    - Trying t[1] = 50 → Error (TypeError).

However:

- You can reassign the whole tuple variable to a new tuple (not modify the existing one).
- If a tuple contains mutable elements (like lists), those can still be modified inside.

2.Creating and accessing elements in a tuple.

1. Creating a Tuple

- Tuples are created by enclosing elements in parentheses ( ), separated by commas.
- Examples:
  - Empty tuple: t = ()
  - Single-element tuple: t = (5,) *(note the comma, otherwise it's just an integer)*
  - Multiple elements: t = (10, 20, 30)
  - Without parentheses (Python allows this): t = 1, 2, 3

Tuples can also hold different data types and even nested tuples:

- t = (1, "hello", 3.5, (10, 20))

2. Accessing Elements in a Tuple

- Tuples use indexing just like lists.
- Indexing starts from 0 for the first element.
    - Example: t[0] → first element, t[2] → third element.
- Negative indexing is also allowed:
    - Example: t[-1] → last element, t[-2] → second last element.

3. Accessing a Range of Elements (Slicing)

- Tuples support slicing using the format:
    - t[start:end:step]
- Example:
    - t[1:3] → elements from index 1 to 2.
    - t[::-1] → reverses the tuple.

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

1. Concatenation (+)

- Tuples can be joined together using the + operator.
- This creates a new tuple with elements of both tuples.
- Example:
    - (1, 2) + (3, 4) → (1, 2, 3, 4)

2. Repetition (*)

- Tuples can be repeated multiple times using the * operator.
- This creates a new tuple with repeated elements.
- Example:
  - (1, 2) * 3 → (1, 2, 1, 2, 1, 2)

3. Membership (in, not in)

- Membership operators are used to check whether an element exists inside a tuple.
- in → returns True if the element exists.
- not in → returns True if the element does not exist.
- Example:
  - 3 in (1, 2, 3) → True
  - 5 not in (1, 2, 3) → True

❖ Accessing Tuples

1.Accessing tuple elements using positive and negative indexing.

1. Positive Indexing

- In positive indexing, counting starts from 0 for the first element.
- Each element in the tuple can be accessed by its position number.
- Example:
  - t = (10, 20, 30, 40, 50)

- t[0] → 10 (first element)
- t[2] → 30 (third element)
- t[4] → 50 (fifth element)

2. Negative Indexing

- In negative indexing, counting starts from -1 for the last element.
- Useful for accessing elements from the end without knowing the length.
- Example:
  - t = (10, 20, 30, 40, 50)
  - t[-1] → 50 (last element)
  - t[-2] → 40 (second last element)
  - t[-5] → 10 (first element, since -5 = index 0)

2. Slicing a tuple to access ranges of elements.

➢ Tuple Slicing

- Slicing allows you to extract a part (sub-tuple) of a tuple without modifying the original tuple.
- Syntax:

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

- start → index to start slicing (inclusive, default 0)
- end → index to stop slicing (exclusive)
- step → interval between elements (default 1)

Examples :

1. Basic slicing:
   - t = (10, 20, 30, 40, 50)
   - t[1:4] → (20, 30, 40) (elements from index 1 to 3)
2. Omitting start or end:
   - t[:3] → (10, 20, 30) (from beginning to index 2)
   - t[2:] → (30, 40, 50) (from index 2 to end)
3. Using step:
   - t[0:5:2] → (10, 30, 50) (every second element)
4. Negative slicing (reversing):
   - t[::-1] → (50, 40, 30, 20, 10) (reverses the tuple)
   - t[3:0:-1] → (40, 30, 20) (elements in reverse from index 3 to 1)

❖ Dictionaries

1. Introduction to dictionaries: key-value pairs.

1. Introduction to Dictionaries

- A dictionary in Python is an unordered collection of items.
- Each item is stored as a key-value pair.
- Dictionaries are mutable, meaning you can change, add, or remove items.
- They are defined using curly braces { }, with a colon : separating keys and values.

Syntax:

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

2. Key-Value Pairs

- Key: A unique identifier used to access the value. Must be immutable (e.g., string, number, or tuple).
- Value: Data associated with the key. Can be of any data type (numbers, strings, lists, tuples, etc.).

Example:

```
student = {
    "name": "Niyati",
    "age": 18,
    "course": "Python"
}
```

- "name" → key, "Niyati" → value

- "age" → key, 18 → value
- "course" → key, "Python" → value

3. Key Points

- Keys must be unique; values can repeat.
- Dictionaries are unordered in Python versions before 3.7; from Python 3.7+, they maintain insertion order.
- Accessing values is done using the key: student["name"] → "Niyati"

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

➢ Accessing Dictionary Elements

- Values are accessed using their keys.
- Syntax: dictionary[key]
- Example:

```
student = {"name": "Niyati", "age": 18}
student["name"]
```

output:

```
Niyati
```

➢ Adding Dictionary Elements

- To add a new key-value pair, simply assign a value to a new key.
- Example:

```
student["course"] = "IT"
```

➢ Updating Dictionary Elements

- To update the value of an existing key, assign a new value to it.
- Example:

```
student.update({"age": 19, "gender": "Female"})
```

➢ Deleting Dictionary Elements

- Using del: Removes a key-value pair by key.

```
del student["course"]
```

- Using .pop(key): Removes a key and returns its value.

```
age = student.pop("age")
```

- Using .clear(): Removes all elements, leaving an empty dictionary.

```
student.clear()
```

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

- 1.keys()

  Returns a view object containing all the keys of the dictionary.

  - Syntax: dictionary.keys()
  - The view object reflects changes made to the dictionary.
  - Example :

    student = {"name": "Niyati", "age": 21}

    student.keys()

2. values()

- Returns a view object containing all the values of the dictionary.
- Syntax: dictionary.values()
- Example :

    student.values()

3. items()

- Returns a view object of key-value pairs as tuples.
- Syntax: dictionary.items()

- Useful for iterating over both keys and values.
- Example:

```
student.items()
```

❖ Working with Dictionaries

1.Iterating over a dictionary using loops.

➢Iterating over Keys

- By default, looping through a dictionary iterates over its keys.
- Syntax:

```
for key in dictionary:
```

- Example:

```
student = {"name": "Niyati", "age": 21}
for key in student:
    print(key, student[key])
```

2. Using keys() Method

- You can explicitly loop over keys using dictionary.keys().
- Equivalent to the default behavior.

- Example:

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

3. Using values() Method

- Loop over all values in the dictionary using values().
- Example:

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

4. Using items() Method

- Loop over key-value pairs simultaneously using items().
- Example :

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

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

➢ Using Loops

- If you have two lists of equal length, you can merge them by using a for loop.
- One list becomes the keys, and the other becomes the values.

Example:

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

values = ["Niyati", 18, "IT"]

dictionary = {}

for i in range(len(keys)):

   dictionary[keys[i]] = values[i]
```

2. Using zip() Function

- zip() pairs elements from two lists into tuples of (key, value).
- Passing it to dict() converts the pairs into a dictionary.

Example:

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

values = ["Niyati", 18, "IT"]

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

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

- **Steps for counting characters in string using dictionary**

Step 1: Initialize an empty dictionary.

Step 2: Traverse each character of the string using a loop.

Step 3: If the character is already present in the dictionary, increment its count.

Step 4 : If the character is not present, add it to the dictionary with count 1.

Step 5 : Finally, print the dictionary which shows characters and their frequencies.

Example:

```
text = "hello world"
fre = {}
for char in text:
   if char in fre:
      fre [char] += 1
   else:
      fre [char] = 1
print(“Character frequencies:”,fre)
```

output:

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

❖ Functions

1. Defining functions in Python.

In Python, a function is a reusable block of code that performs a specific task. You define a function using the def keyword.

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

1. Function Without Parameters and Without Return Value

This type of function does not take any input and does not return any value.

```
def greet():
    print("Hello")
```

2. Function With Parameters and Without Return Value

This function takes input parameters but does not return any value.

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

3. Function Without Parameters and With Return Value

This function does not take input parameters but returns a value.

```
def get_number():
    return 10
```

4. Function With Parameters and With Return Value

This function takes input parameters and returns a value.

```
def multiply(a, b):
    return a * b
```

3. Anonymous functions (lambda functions).

An anonymous function in Python is a small function that is defined without a name. It is also called a lambda function and is created using the lambda keyword.

Definition:
A lambda function is a one-line function that can take any number of arguments but can have only one expression.

Syntax

```
lambda arguments: expression
```

Example:

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

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

❖ Modules

1.Introduction to Python modules and importing modules

A module in Python is a file that contains Python code such as functions, variables, and classes. Modules help in organizing code and promoting reusability.

Definition:
A Python module is a file with a .py extension that can be imported and used in another Python program.

Importing Modules in Python:

1. Import Entire Module:

```
import math

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

2. Import Specific Members:

```
from math import sqrt

print(sqrt(25))
```

2. Standard library modules: math, random.

Python provides a standard library that contains many built-in modules. These modules are ready to use and do not require installation.

1. math Module

The math module is used to perform mathematical operations such as square root, power, trigonometry, etc.

Common Functions of math Module:

```
import math

math.sqrt(16)

math.pow(2, 3)

math.factorial(5)
```

2. random Module

The random module is used to generate random numbers and perform random operations.

Common Functions of random Module

```
import random

random.randint(1, 10)    # Random integer between 1 and 10

random.random()          # Random float between 0 and 1

random.choice([1, 2, 3])  # Random element from a list

random.shuffle(list)     # Shuffle elements of a list
```

3. Creating custom modules.

A custom module is a Python file created by the user that contains functions, variables, or classes, which can be reused in other Python programs.

Steps to Create a Custom Module

1. Create a Python file
   Example: mymodule.py

```
def greet(name):
    return f"Hello, {name}"
def add(a, b):
    return a + b
```

2. Import the Module in Another File

   Example: main.py

```
import mymodule
print(mymodule.greet("Niyuu"))
print(mymodule.add(5, 3))
```

3 .Import Specific Functions

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