Module - 7: Python – Collections, functions and Modules

1. Accessing List

Q1. Understanding how to create and access elements in a list.

Ans:

- ➢ Creating a list:

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

- ➢ Accessing elements in a list:
- Use square brackets [] with the index of the item.

```
print(fruits[0])     # Output: apple
print(fruits[1])    # Output: banana
print(fruits[2])   # Output: cherry
```

Q2. Indexing in lists (positive and negative indexing).

- **Positive indexing** starts from 0 (left to right).
- **Negative indexing** starts from -1 (right to left).

Example:

```
fruits = ["apple", "banana", "cherry", "date"]
# Positive indexing
print(fruits[0])  # apple
print(fruits[2])  # cherry
# Negative indexing
print(fruits[-1])  # date
print(fruits[-3])  # banana
```

Q3. Slicing a list: accessing a range of elements.

Syntax: list[start:stop]

- Includes elements from start to stop - 1.

Example:

```
fruits = ["apple", "banana", "cherry", "date", "elderberry"]
# Slice from index
print(fruits[1:4])      # ['banana', 'cherry', 'date']
```

```
#Start or end to slice from/to beginning or end
print(fruits[:3])        # ['apple', 'banana', 'cherry']
print(fruits[2:])        # ['cherry', 'date', 'elderberry']
# Slicing with negative indexes
print(fruits[-4:-1])   # ['banana', 'cherry', 'date']
```

2. List Operations

Q4. Common list operations: concatenation, repetition, membership.

➢ Following are operations you can perform on lists using simple syntax.

i. Concatenation (+): Join two or more lists together.

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

ii. Repetition (*): Repeat the elements of a list multiple times.

```
nums = [0, 1]
repeated = nums * 3
print(repeated)      # [0, 1, 0, 1, 0, 1]
```

iii. Membership (in, not in): Check if an element exists in a list.

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

Q5. Understanding list methods like append(), insert(), remove(), pop().

i. append(item): Adds an item to the end of the list.

```
colors = ["red", "blue"]
colors.append("green")
print(colors)              # ['red', 'blue', 'green']
```

ii. insert(index, item): Inserts an item at a specific position.

```
colors.insert(1, "yellow")
print(colors)              # ['red', 'yellow', 'blue', 'green']
```

iii. remove(item): Removes the first occurrence of the item.

```
colors.remove("blue")

print(colors)             # ['red', 'yellow', 'green']
```

iv. pop(index): Removes and returns the item at the given index. If no index is given, removes the last item.

```
last_color = colors.pop()

print(last_color)          # 'green'

print(colors)               # ['red', 'yellow']

second_color = colors.pop(1)

print(second_color)     # 'yellow'

print(colors)              # ['red']
```

3. Working with Lists

Q6. Iterating over a list using loops.

i.Using for loop:

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

for fruit in fruits:

   print(fruit)
```

ii.Using for loop with index (range)

```
for i in range(len(fruits)):

   print(f"{i}: {fruits[i]}")
```

iii.Using while loop:

```
i = 0

while i < len(fruits):

   print(fruits[i])

   i += 1
```

Q7. Sorting and reversing a list using sort(), sorted(), and reverse().

i.sort() — Sorts the list in place**.** Modifies the original list.

```
list = [4, 2, 7, 1]

list.sort()

print(list)       # [1, 2, 4, 7]
```

ii. sorted() — Returns a new sorted list and the original list remains unchanged.

```
list = [4, 2, 7, 1]

sorted_list = sorted(list)

print(sorted_list)      # [1, 2, 4, 7]

print(list)             # [4, 2, 7, 1]
```

iii.reverse() — reverses the list in place

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

fruits.reverse()

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

iv.Reverse using slicing:

```
reversed_list = fruits[::-1]

print(reversed_list)            # ['apple', 'banana', 'cherry']
```

Q8. Basic list manipulations: addition, deletion, updating, and slicing.

i.Addition

- append(item) – Add to end
- insert(index, item) – Add at position
- + – Concatenate lists

Example:

```
colors = ["red", "green"]

colors.append("blue")

colors.insert(1, "yellow")

print(colors)  # ['red', 'yellow', 'green', 'blue']
```

ii.Deletion

- remove(item) – Delete by value
- pop(index) – Delete by index
- del list[index] – Delete by index
- clear() – Remove all items

Example:

```
del colors[0]       # removes 'red'

colors.remove("green")

print(colors)       # ['yellow', 'blue']
```

iii.Updating: Changes an item by assigning a new value using its index.

Example:

```
colors[1] = "purple"

print(colors)  # ['yellow', 'purple']
```

iv.Slicing: Syntax [start:stop:step]

Example:

```
numbers = [0, 1, 2, 3, 4, 5]

print(numbers[1:4])           # [1, 2, 3]

print(numbers[:3])            # [0, 1, 2]

print(numbers[::2])           # [0, 2, 4]
```

4. Tuple

Q9. Introduction to tuples, immutability.

- What is a tuple?
  - A **tuple** is an ordered collection of items, just like a list.
  - Unlike lists, **tuples cannot be changed** (immutable).
- Why use tuples?
  - They're faster than lists.
  - They're useful for fixed data (e.g., coordinates, RGB values).
  - They're hashable and can be used as dictionary keys.

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

print(my_tuple)             #(1,2,2,3)
```

- Immutability using example

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

# my_tuple[0] = 100          # This will raise a TypeError
```

Q10. Creating and accessing elements in a tuple.

i.Creating tuples:

```
t1 = (1, 2, 3)

t2 = ("apple", "banana", "cherry")

t3 = ()                # Empty tuple

t4 = (5,)              # Single-element tuple (comma is required!)
```

Accessing tuple elements: It is same as lists using indexing and slicing.

```
colors = ("red", "green", "blue")
print(colors[0])         # red
print(colors[-1])        # blue
print(colors[1:3])       # ('green', 'blue')
```

Q11. Basic operations with tuples: concatenation, repetition, membership.

i. Concatenation (+)

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

ii. Repetition (*)

```
t = ("A",)
print(t * 3)         # ('A', 'A', 'A')
```

iii. Membership (in, not in)

```
colors = ("red", "green", "blue")
print("green" in colors)            # True
print("yellow" not in colors)       # True
```

5. Accessing Tuples

Q12. Accessing tuple elements using positive and negative indexing.

i.Positive Indexing

- Starts from **0** (left to right)

Example:

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

ii.Negative Indexing

- Starts from **-1** (right to left)

Example:

```
print(fruits[-1])  # date
print(fruits[-3])  # banana
```

Diagram:

```
Index:      0        1          2        3

          'apple' 'banana' 'cherry' 'date'

              -4       -3          -2       -1
```

Q13. Slicing a tuple to access ranges of elements.

- **Syntax: tuple[start:stop:step]**
- Returns a **new tuple** from start to stop - 1.

Example:

```
colors = ("red", "green", "blue", "yellow", "purple")

# Slice from index 1 to 3

print(colors[1:4])  # ('green', 'blue', 'yellow')

# Slice from start to index 2

print(colors[:3])   # ('red', 'green', 'blue')

# Slice from index 2 to end

print(colors[2:])   # ('blue', 'yellow', 'purple')

# Slice with step

print(colors[::2])  # ('red', 'blue', 'purple')

# Reverse the tuple

print(colors[::-1])  # ('purple', 'yellow', 'blue', 'green', 'red')
```

6. Dictionaries

Q14. Introduction to dictionaries: key-value pairs.

- What is a dictionary?
- A dictionary is an unordered, mutable collection of items. Each item is stored as a key-value pair.
- Keys must be unique and immutable (like strings, numbers, or tuples).
- Values can be of any data type.

Syntax:

```
my_dict = {   "name": "Alice",

                    "age": 25,
                "city": "New York"   }
```

Q15. Accessing, adding, updating, and deleting dictionary elements.

i.Accessing values

```
print(my_dict["name"])      # Alice
# using get() method
print(my_dict.get("age"))     # 25
```

ii.Adding a new key-value pair

```
my_dict["email"] = alice@example.com
print (my_dict)
```

iii.Updating a value

```
my_dict["age"] = 26
```

iv.Deleting an item

```
del my_dict["city"]          # Removes the key 'city'
my_dict.pop("age")          # Also removes and returns value of 'age'
my_dict.clear()             # Removes all items from the dictionary
```

Q16. Dictionary methods like keys(), values(), and items().

- These are useful for looping over dictionaries or inspecting their contents. These methods help you access different parts of a dictionary. These views can be converted to lists if needed.

- **keys():** Returns a view of all keys.
  ```
  student = {"name": "Alex", "age": 20, "grade": "A"}
  print(student.keys())         # dict_keys(['name', 'age', 'grade'])
  ```

- **values():** Returns a view of all values.
  ```
  print(student.values())      # dict_values(['Alex', 20, 'A'])
  ```

- **items():** Returns a view of all key-value pairs (as tuples).
  ```
  print(student.items())       # dict_items([('name', 'Alex'), ('age', 20), ('grade', 'A')])
  ```

- **Looping with items()**
  ```
  for key, value in my_dict.items():
  print(f"{key}: {value}")
  ```

7. Working with Dictionaries

Q17. Iterating over a dictionary using loops.

You can iterate a dictionary through:

- Keys
- Values
- Key-value pairs

i. Iterating over keys (default)

```
person = {"name": "Alice", "age": 25, "city": "New York"}
for key in person:
    print(key, "->", person[key])
```

ii. Iterating over .items() (key-value pairs)

```
for key, value in person.items():
    print(f"{key}: {value}")
```

iii. Iterating over values

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

Q18. Merging two lists into a dictionary using loops or zip().

Here, in both the methods the keys and values are of the same length.

i.Using zip()

```
keys = ["name", "age", "city"]
values = ["Alice", 25, "Surat"]
merged = dict(zip(keys, values))
print(merged)
# {'name': 'Alice', 'age': 25, 'city': 'Surat'}
```

ii.Using a loop

```
merged = {}
for i in range(len(keys)):
    merged[keys[i]] = values[i]
print(merged)
```

Q19. Counting occurrences of characters in a string using dictionaries.

To count how often each character appears in a string, use a dictionary.

```
text = "hello world"
char_count = {}
```

```
for char in text:
    if char in char_count:
        char_count[char] += 1
    else:
        char_count[char] = 1
print(char_count)                     # {'h': 1, 'e': 1, 'l': 3, 'o': 2, ' ': 1, 'w': 1, 'r': 1, 'd': 1}
```

8. Functions

Q20. Defining a function in Python.

- A function is a reusable block of code that can performs a specific task.

Syntax:

```
def greet():
    print("Hello!")
#Calling the function:
greet()
```

Q21. Different types of functions: with/without parameters, with/without return values.

i. Without parameters, no return type:

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

ii. With parameters, no return type:

```
def greet_user(name):
    print(f"Hello, {name}!")
```

iii. With parameters, with return value:

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

iv. Without parameters, with return:

```
def get_name():
    return "Alice"
```

Q22. Anonymous functions (lambda functions).

- Lambda functions are small, one-line anonymous functions.
- **Syntax:** lambda arguments: expression.
- Use them when a short function is needed for a short time (e.g. with map(), filter(), or sorted()).

Example:

```
add = lambda x, y: x + y

print(add(3, 4))          # Output: 7
```

9. Modules

Q23. Introduction to Python modules and importing modules.

- A module is a file containing Python code (functions, variables, classes).

i.Importing a module:

```
import math

import random
```

ii.Importing a specific item:

```
from math import sqrt
```

Q24. Standard library modules: math, random.

i. math module:

```
import math

print(math.sqrt(25))           # 5.0

print(math.pi)                 # 3.141592653589793

print(math.factorial(5))       # 120
```

ii. random module:

```
import random

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

print(random.choice(['a', 'b']))        # Random choice from a list
```

Q25. Creating custom modules.

- You can create your own module by saving a .py file with functions. Ie. Here math_utils.py.
- Then use it in another file.
- Make sure both files are in the same directory or properly configured in your project.

Example:

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

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

import math_utils
print(math_utils.add(2, 3))      # 5
print(math_utils.multiply(4, 5)) # 20
```