Module 14
Name:- Rahul Zapadiya
Python – Collections, functions and Modules

Lists in Python – Theory

1. Creating and Accessing Elements in a List

• A list is an ordered, mutable (changeable) collection of items in Python.

• Lists can store elements of different data types (integers, strings, floats, even other lists).

• Lists are created using square brackets [] or the list() constructor.

Examples:

Creating lists

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

mixed = [1, "Hello", 3.14, True]

nested = [1, [2, 3], 4]

#Accessing elements

print(numbers[0]) # First element → 1

print(mixed[1]) # Second element → Hello

print(nested[1]) # Inner list → [2, 3]

2.Indexing in Lists

• Positive Indexing: Index starts from 0 for the first element.

• Negative Indexing: Index starts from -1 for the last element, -2 for second last, and so on.

Example:

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

#Positive indexing

print(fruits[0]) # apple

print(fruits[2]) # cherry

#Negative indexing

print(fruits[-1]) # cherry

print(fruits[-2]) # banana

3.Slicing a List

•   Slicing allows you to access a range of elements from a list.

•   Syntax: list[start:end:step]

o   start → index to begin (inclusive)

o   end → index to stop (exclusive)

o   step → skip elements (default is 1)

Example:

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

#Slicing examples

print(numbers[1:4]) # [20, 30, 40] → from index 1 to 3

print(numbers[:3]) # [10, 20, 30] → from start to index 2

print(numbers[3:]) # [40, 50, 60] → from index 3 to end

print(numbers[::2]) # [10, 30, 50] → every 2nd element

print(numbers[::-1]) # [60, 50, 40, 30, 20, 10] → reversed list

List Operations in Python – Theory

1. Common List Operations

Python allows several basic operations on lists:

a) Concatenation

•   Combines two or more lists into a single list using the + operator.

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

result = list1 + list2

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

b) Repetition

•   Repeats the elements of a list using the * operator.

```python
list1 = ["a", "b"]
result = list1 * 3
print(result) # ['a', 'b', 'a', 'b', 'a', 'b']
```

c) Membership

•   Checks if an element exists in the list using the in or not in operators.

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


2.Common List Methods

a)append()

•   Adds a single element to the end of the list.

```python
numbers = [1, 2, 3]
numbers.append(4)
print(numbers) # [1, 2, 3, 4]
```

b) insert()

•   Inserts an element at a specific index.

```python
numbers = [1, 3, 4]
numbers.insert(1, 2)
print(numbers) # [1, 2, 3, 4]
```

c) remove()

•   Removes the first occurrence of a specific element.

```python
numbers = [1, 2, 3, 2]
numbers.remove(2)
print(numbers) # [1, 3, 2]
```

d) pop()

•   Removes and returns an element at a given index (default is the last element).

```python
numbers = [1, 2, 3]
last_item = numbers.pop()
```

```
print(last_item) # 3
print(numbers) # [1, 2]


second_item = numbers.pop(0)
print(second_item) # 1
print(numbers)     # [2]
```

Working with Lists – Theory

1. Iterating Over a List Using Loops

•   You can loop through each element in a list using for or while loops.

Using a for loop:

```
fruits = ["apple", "banana", "cherry"]
for fruit in fruits:
    print(fruit)
```

Using a while loop:

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


2.Sorting and Reversing a List

a)sort()

•   Sorts the list in-place (changes the original list).

```
numbers = [4, 2, 8, 1]
numbers.sort()
print(numbers) # [1, 2, 4, 8]
```

•   To sort in descending order:

```
numbers.sort(reverse=True)
print(numbers) # [8, 4, 2, 1]
```

b) sorted()

- Returns a new sorted list without changing the original list.

numbers = [4, 2, 8, 1] new_list = sorted(numbers) print(new_list) # [1, 2, 4, 8] print(numbers) # [4, 2, 8, 1] (unchanged)

c) reverse()

- Reverses the order of elements in-place.

numbers = [1, 2, 3]

numbers.reverse()

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

3.Basic List Manipulations

a)Addition

- append() → Adds an element to the end.
- insert() → Adds an element at a specific position.
- extend() → Adds multiple elements from another list.

list1 = [1, 2]

list1.append(3)   # [1, 2, 3]

list1.insert(1, 5)   # [1, 5, 2, 3]

list1.extend([6, 7]) # [1, 5, 2, 3, 6, 7]

b) Deletion

- remove() → Removes the first occurrence of an element.
- pop() → Removes by index (or last if index not given).
- del → Deletes an element or slice.

list1 = [1, 2, 3, 4]

list1.remove(2)   # [1, 3, 4]

list1.pop(0)   # [3, 4]

del list1[1]   # [3]

c) Updating

- Change a value at a specific index.

list1 = [1, 2, 3]

list1[1] = 99

print(list1) # [1, 99, 3]

d) Slicing

•   Extracting a part of the 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]

Tuple – Theory

1. Introduction to Tuples & Immutability

•   Tuple is a built-in Python data type used to store multiple items in a single variable.

•   It is similar to a list but immutable, meaning once created, its elements cannot be changed, added, or removed.

•   Tuples are defined using parentheses ().

•   Because tuples are immutable, they are faster than lists and can be used as dictionary keys.

Example:

t=(1, 2, 3)

#t[0] = 10 # Error: Tuples are immutable

2.Creating and Accessing Elements in a Tuple

•   Creating a Tuple:

tuple1 = (10, 20, 30)   # Normal tuple

tuple2 = ("apple", "banana") # Tuple with strings

tuple3 = (1, "apple", 3.5) # Mixed data types

tuple4 = (10,)   # Single element tuple (comma is necessary)

•   Accessing Elements:

my_tuple   =   (10,   20,   30,   40)

print(my_tuple[0]) # First element → 10

print(my_tuple[-1]) # Last element → 40

3. Basic Operations with Tuples

a) Concatenation

• 		Join two tuples together using +.

t1 = (1, 2) t2 = (3, 4)

t3 = t1 + t2 print(t3) #

(1, 	2, 	3, 	4) 	b)

Repetition

• 		Repeat a tuple multiple times using *.

t = (5, 6)

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

c) Membership

• 		Check if an element exists in a tuple using in or not in.

t = (1, 2, 3)

print(2 in t) 	# True

print(5 not in t) # True

Accessing Tuples – Theory

1. Accessing Tuple Elements with Positive Indexing

• 		Positive indexing starts from 0 for the first element.

• 		You can directly use the index inside square brackets [] to get a specific element.

Example:

t = (10, 20, 30, 40, 50)

print(t[0]) # 10 → First element

print(t[2]) # 30 → Third element

2.Accessing Tuple Elements with Negative Indexing

• 		Negative indexing starts from -1 for the last element and moves backward.

• 		This is useful when accessing elements from the end without knowing the tuple's length.

Example:

t = (10, 20, 30, 40, 50)

print(t[-1]) # 50 → Last element

print(t[-3]) # 30 → Third element from the end

3.Slicing a Tuple

•   Slicing allows you to access a range of elements.

•   Syntax:

•   tuple[start:end:step]

o   start → index where slicing begins (default: 0)

o   end → index where slicing ends (exclusive)

o   step → gap between elements (default: 1)

Example:

t= (10, 20, 30, 40, 50)

print(t[1:4]) # (20, 30, 40) → From index 1 to 3

print(t[:3]) # (10, 20, 30) → From start to index 2

print(t[2:]) # (30, 40, 50) → From index 2 to end

print(t[::2]) # (10, 30, 50) → Every 2nd element

print(t[::-1]) # (50, 40, 30, 20, 10) → Reversed tuple

6.Dictionaries – Theory

1.Introduction to Dictionaries

•   A dictionary is a collection of key-value pairs.

•   Keys must be unique and immutable (strings, numbers, or tuples).

•   Values can be of any data type (mutable or immutable).

•   Dictionaries are unordered (before Python 3.7) but insertion-ordered from Python 3.7+.

Example:

my_dict = {

   "name": "rahul",

   "age": 21, "city":

   "Ahmedabad"

}

Here:

- •   "name", "age", "city" are keys.

     "rahul", 21, "Ahmedabad" are

     values.

2.Accessing Dictionary Elements

-     Using keys inside square brackets []:

```
print(my_dict["name"]) # rahul
```

-     Using .get() method (avoids errors if key doesn't exist):

```
print(my_dict.get("age")) # 21

print(my_dict.get("salary", "Not Found")) # Default value if key is missing
```

3.Adding Elements

-     Simply assign a value to a new key:

```
my_dict["salary"] = 50000
```

4.Updating Elements

-     Assign a new value to an existing key:

```
my_dict["age"] = 22
```

5.Deleting Elements

-     Using del:

```
delmy_dict["city"]
```

-     Using .pop():

```
my_dict.pop("salary")
```

-     Using .clear() to remove all elements:

```
my_dict.clear()
```

6.Dictionary Methods

-     .keys() → Returns all keys:

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

- .values() → Returns all values:

print(my_dict.values()) # dict_values(['rahul', 21])

- .items() → Returns all key-value pairs as tuples:

print(my_dict.items()) # dict_items([('name', 'rahul'), ('age', 21)])

7.Working with Dictionaries – Theory

1.Iterating Over a Dictionary

You can loop through:

- Keys only (default iteration):

```
my_dict = {"name": "rahul", "age": 21}

forkey in my_dict:

    print(key, my_dict[key])

#Output:

#name rahul

#age 21
```

- Keys explicitly:

```
forkey in my_dict.keys():

    print(key)
```

- Values:

```
forvalue in my_dict.values():

    print(value)
```

- Key-Value pairs:

```
forkey, value in my_dict.items():

    print(f"{key} → {value}")
```


2.Merging Two Lists into a Dictionary Youcan combine a list of keys and a list of values into a dictionary. Using zip(): keys = ["name", "age", "city"] values = ["rahul", 21, "Ahmedabad"] merged_dict = dict(zip(keys, values))

print(merged_dict) # Output: {'name': 'rahul', 'age': 21,

'city': 'Ahmedabad'} Using loops: keys = ["name", "age",

"city"] values = ["rahul", 21, "Ahmedabad"] merged_dict =

{} for i in range(len(keys)):

```
    merged_dict[keys[i]] = values[i]
print(merged_dict)
```

3.Counting Occurrences of Characters in a String

Dictionaries are perfect for counting frequency.

Example:

```
text = "programming"
char_count = {}
for char in text:
    char_count[char] = char_count.get(char, 0) + 1
print(char_count)
#Output: {'p': 1, 'r': 2, 'o': 1, 'g': 2, 'a': 1, 'm': 2, 'i': 1, 'n': 1}
```

8.Functions – Theory

1.Defining Functions in Python

Afunction is a block of reusable code that performs a specific task.

Syntax:

```
def function_name(parameters):
    """Optional docstring describing the function"""
    # function body
    return value # optional
```

Example:

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

```
greet() # Calling the function
```

2. Types of Functions

a) Without Parameters and Without Return Value

```
defsay_hello():
    print("Hello, World!")
say_hello()
```

b) With Parameters and Without Return Value

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

c) Without Parameters but With Return Value

```
def get_pi():
    return 3.14159
print(get_pi())
```

d) With Parameters and With Return Value

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

3. Anonymous Functions (Lambda Functions)

Alambda function is a small, one-line, anonymous function in Python.

Syntax:

```
lambda arguments: expression
```

Example:

```
square = lambda x: x ** 2
print(square(5)) # Output: 25


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

Lambdas are often used with functions like map(), filter(), and sorted().

9.Modules – Theory

1.Introduction to Python Modules

•   A module in Python is simply a file containing Python code (functions, classes, or variables) that can be reused in other programs.

•   Helps in code reusability and better organization.

•   Python comes with a large collection of built-in modules called the Standard Library.

Importing a module:

import module_name

Importing specific items from a module:

from module_name import function_name

2.Standard Library Modules

a)math Module

•   Provides mathematical functions and constants.

Example:

import math

print(math.sqrt(16)) # 4.0

print(math.pi)   # 3.141592653589793

print(math.factorial(5)) # 120

b) random Module

•   Used for generating random numbers and selections.

Example:

import random

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

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

3.Creating Custom Modules

Steps:

1.   Create a Python file (e.g., mymodule.py) containing functions or variables.

2.   Import it into another program using import.

Example:

mymodule.py

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

```python
main.py import mymodule
print(mymodule.greet("Rahul"))
```