Modual-2 Python Assignment

1. How functional programming works in Python.
   Functional programming in Python is a programming paradigm that treats computation as the evaluation of mathematical functions and avoids changing state and mutable data. Python supports functional programming features, even though it is not a purely functional language.

   User-define Function
   Lambda Functions

   Map, Filter, and Reduce(Inbuilt Function)
2. **Using map(), reduce(), and filter() functions for processing data.**
   map()

```
numbers = [1, 2, 3, 4]
squared = list(map(lambda x: x ** 2, numbers))
print(squared)
```

   reduce()

```
from functools import reduce
numbers = [1, 2, 3, 4]
product = reduce(lambda x, y: x * y, numbers)
print(product)
```

   Filter()

```
numbers = [1, 2, 3, 4, 5]
evens = list(filter(lambda x: x % 2 == 0, numbers))
print(evens)  # Output: [2, 4]
```

3. Introduction to closures and decorators.
   A closure is a function that remembers the values from its enclosing scope even after the scope has finished executing. Closures are used to create function factories, encapsulation, and maintain state.
   A decorator is a special type of closure that allows you to modify the behavior of a function **without modifying its actual code**. Decorators are widely used for logging, authentication, and enforcing access control.

4. Understanding how to create and access elements in a list
   You can create a list using square brackets []:

```
numbers = [1, 2, 3, 4, 5]
fruits = ["apple", "banana", "cherry"]
mixed = [1, "hello", 3.5, True]
```

   Lists are **zero-indexed**, meaning the first element has index [0].

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

5. Indexing in lists (positive and negative indexing).
   1. Positive Indexing

- Positive indexing starts from 0 for the first element and increases by 1 for each subsequent element.

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

```
print(fruits[0])

print(fruits[2])

print(fruits[4])
```

2. Negative Indexing

- Negative indexing starts from -1 for the last element and decreases by 1 as you move left.

```
print(fruits[-1])

print(fruits[-2])

print(fruits[-5])
```

6. Slicing a list: accessing a range of elements

Slicing in Python allows you to access a **subset of elements** from a list using the syntax:

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

- **start** – The index where slicing begins
- **end** – The index where slicing stops
- **step** – The interval between elements

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

1. Concatenation (+ Operator)

Concatenation allows you to **combine** two or more lists into a single list.

Example:

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

list2 = [4, 5, 6]

combined = list1 + list2

print(combined)
```

2. Repetition (* Operator)

Repetition allows you to **repeat** a list multiple times.

Example:

```
nums = [1, 2, 3] * 3

print(nums)

zeros = [0] * 5

print(zeros)  # Output: [0, 0, 0, 0, 0]
```

3. Membership (in and not in Operators)

Membership operators allow you to check if a value **exists** in a list.

Example:

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


print("banana" in fruits)

print("grape" not in fruits)

if "apple" in fruits:

   print("Apple is available!")
```

8. Understanding list methods like append(), insert(), remove(), pop().
   Python provides built-in **list methods** for adding, removing, and modifying elements. Here, we'll explore some of the most commonly used methods:

1. **append()** → Adds an element to the end
   list.append(element)

2. **insert()** → Inserts an element at a specific position
   list.insert(index, element)

3. **remove()** → Removes the first occurrence of a specific value
   list.remove(value)

4. **pop()** → Removes and returns an element by index
   list.pop(index)

9. Iterating over a list using loops

For Loop

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

While Loops

```
numbers = [10, 20, 30, 40]
index = 0
while index < len(numbers):
   print(numbers[index])
   index += 1
```

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

1. Using sort() (Modifies the List)

The sort() method **modifies the list itself** and doesn't return anything.

Example:-

```
numbers = [5, 2, 9, 1, 5, 6]
```

numbers.sort()

print(numbers)

- sort() changes the original list.
- Sorting is done in **ascending order** by default.

Sorting in Descending Order

Use reverse=True to sort in **descending** order.

numbers.sort(reverse=True)

print(numbers)

2. Using sorted() (Creates a New List)

The sorted() function returns a **new sorted list** without modifying the original.

Example:-

numbers = [5, 2, 9, 1, 5, 6]

sorted_numbers = sorted(numbers)

print(sorted_numbers)

print(numbers)

Sorting in Descending Order

desc_sorted = sorted(numbers, reverse=True)

print(desc_sorted)

3. Using reverse() (Reverses In-Place)

The reverse() method **reverses the list in place** (modifies original list).

Example:-

letters = ["a", "b", "c", "d"]

letters.reverse()

print(letters)

4. Reversing Using Slicing ([::-1])

You can create a **reversed copy** using slicing.

Example:-

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

reversed_numbers = numbers[::-1]

print(reversed_numbers)

print(numbers)

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

Operation	Method	Example	Output
Add Element (End)	append(x)	[1, 2].append(3)	[1, 2, 3]
Add at Index	insert(i, x)	[1, 3].insert(1, 2)	[1, 2, 3]
Add Multiple Elements	extend([x, y])	[1, 2].extend([3, 4])	[1, 2, 3, 4]
Remove by Value	remove(x)	["a", "b"].remove("a")	["b"]
Remove by Index	pop(i)	[1, 2, 3].pop(1)	[1, 3]
Delete by Index	del list[i]	del list[0]	Removes element at index 0
Delete All	clear()	[1, 2].clear()	[]
Update Element	list[i] = x	list[1] = 10	Updates index 1
Slicing	list[start:stop:step]	[0, 1, 2, 3][1:3]	[1, 2]

12. Introduction to tuples, immutability

Feature	Tuples	Lists
Mutability	❌ Immutable	✅ Mutable
Performance	✅ Faster	❌ Slower (when modified)
Memory Usage	✅ Less memory	❌ More memory
Hashable (for dict keys)	✅ Yes	❌ No

13. Creating and accessing elements in a tuple

1. Creating a Tuple

Tuples are defined using **parentheses ()** or simply by separating values with **commas**.

Examples:

```
empty_tuple = ()

# Tuple with multiple elements
numbers = (1, 2, 3, 4, 5)

# Tuple with different data types
mixed = (1, "hello", 3.5, True)

# Tuple without parentheses (implicit packing)
colors = "red", "green", "blue"

print(numbers)
print(colors)

print(type(single_element))
print(type(not_a_tuple))
```

2. Accessing Elements in a Tuple

You can **access elements** using **indexing** (positive or negative) and **slicing**.

Indexing (Positive and Negative)

- **Positive Indexing** → Starts from 0 (left to right).
- **Negative Indexing** → Starts from -1 (right to left).

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

print(fruits[0])  # Output: apple
print(fruits[1])
print(fruits[-1])
```

Slicing (Extracting a Range)

Syntax: tuple[start:stop:step]

- **start** → Beginning index (default: 0)
- **stop** → End index (not included)

- **step** → Skip elements (default: 1)

numbers = (10, 20, 30, 40, 50, 60)

print(numbers[1:4])
print(numbers[:3])
print(numbers[2:])
print(numbers[::2])
print(numbers[::-1])

3. Accessing Tuple Elements Using Loops
Using a for Loop
fruits = ("apple", "banana", "cherry")

for fruit in fruits:
    print(fruit)

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

Operation	Symbol	Example	Output
Concatenation	+	(1, 2) + (3, 4)	(1, 2, 3, 4)
Repetition	*	("A",) * 3	('A', 'A', 'A')
Membership	in	3 in (1, 2, 3)	True
Membership	not in	4 not in (1, 2, 3)	True

15. Accessing tuple elements using positive and negative indexing

1. Positive Indexing (Left to Right)
- The first element starts at **index 0**.
- The second element is at **index 1**, and so on.

2. Negative Indexing (Right to Left)
- The last element starts at **index -1**.
- The second last element is at **index -2**, and so on.

3. Accessing Elements Using a Loop
Instead of manual indexing, you can iterate over a tuple using a loop.

16. Slicing a tuple to access ranges of elements.

Slicing Expression	Result
tuple[start:stop]	Extracts elements from start to stop-1
tuple[:stop]	Extracts elements from the beginning to stop-1
tuple[start:]	Extracts elements from start to the end
tuple[::step]	Extracts elements with a step size
tuple[::-1]	Reverses the tuple

17. Introduction to dictionaries: key-value pairs.

```
student = {
    "name": "Alice",
    "age": 25,
    "grade": "A"
}

print(student)
{'name': 'Alice', 'age': 25, 'grade': 'A'}
```

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

Operation	Syntax	Example
Access Value	dict[key]	student["name"]
Access with get()	dict.get(key, default)	student.get("age", 0)
Add New Key	dict[key] = value	student["city"] = "NY"
Update Key	dict[key] = new_value	student["age"] = 26
Update Multiple	dict.update({...})	student.update
Delete Key	del dict[key]	del student["age"]
Delete and Return	dict.pop(key)	student.pop("name")
Delete Last Item	dict.popitem()	student.popitem()
Clear All	dict.clear()	student.clear

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

Method	Description	Example	Output
keys()	Returns all dictionary keys	student.keys()	dict_keys(['name', 'age', 'grade'])
values()	Returns all dictionary values	student.values()	dict_values(['Alice', 25, 'A'])
items()	Returns key-value pairs as tuples	student.items()	dict_items([('name', 'Alice'), ('age', 25), ('grade', 'A')])

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

```
keys = ["name", "age", "city"]
values = ["Alice", 25, "New York"]
dictionary = {}
for i in range(len(keys)):
    dictionary[keys[i]] = values[i]
print(dictionary)
```

21. Counting occurrences of characters in a string using dictionaries

Method	Code Example	Notes
For Loop	if char in dict: dict[char] += 1	Explicit checks needed
Using get()	dict[char] = dict.get(char, 0) + 1	Cleaner approach
Using Counter	Counter(string)	Most efficient

22. Defining functions in Python.
A **function** in Python is a block of reusable code that performs a specific task.
Functions help in making code **modular, reusable, and easy to debug**.

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

Function Type	Example	Notes
Without Parameters, Without Return	def greet(): print("Hi")	Just prints output
With Parameters, Without Return	def greet(name):	Uses input but no return
Without Parameters, With Return	def get_message():	Returns a value
With Parameters, With Return	def add(a, b): return a + b	Takes input & returns result

24. Introduction to Python modules and importing modules

Import entire module	import module_name	Access using module_name.function()
Import specific function	from module_name import func	Call function directly
Rename module	import module_name as alias	Use alias.function()
Import all (*)	from module_name import *	Avoid due to potential conflicts

25. Standard library modules: math, random.

Module	Purpose	Example Usage
math	Mathematical operations	math.sqrt(16), math.pi, math.ceil(4.2)
random	Random number generation	random.randint(1, 100), random.choice(['a', 'b'])

26. Opening files in different modes ('r', 'w', 'a', 'r+', 'w+')

Mode	Description	Behavior
'r'	Read mode	Opens file for reading (default). **Error if file does not exist.**
'w'	Write mode	Creates a new file or overwrites if it exists.
'a'	Append mode	Opens file for appending, does not delete existing content.
'r+'	Read & Write	Reads and updates a file. **Error if file does not exist.**
'w+'	Write & Read	Creates a new file or overwrites if it exists.

27. Using the open() function to create and access files
    - `"filename.txt"` → Name of the file
    - `mode` → Specifies how the file should be opened (e.g., read, write, append)

28. Closing files using close().
    When working with files in Python, it's important to **close the file** after performing operations like reading or writing. This **frees up system resources** and ensures that all changes are properly saved.

29. Reading from a file using read(), readline(), readlines()

    Python provides multiple ways to read data from a file:

    `read()` → Reads the entire file.
    `readline()` → Reads a single line at a time.
    `readlines()` → Reads all lines and returns them as a list.

30. Writing to a file using write() and writelines().

1. **Writing Singal line String**

```
with open("example.txt", "w") as file:
    file.write("Hello, Python!\n")
    file.write("Writing to a file is easy.")
```

2. Writing Multiple Lines Using writelines()

```
lines = ["Hello, World!\n", "Python is great!\n", "File
handling is fun!\n"]

with open("example.txt", "w") as file:

    file.writelines(lines)
```

31. Introduction to exceptions and how to handle them using try, except, and finally.

`try`	Contains the code that may raise an exception
`except`	Handles the exception
`else`	Runs only if no exception occurs
`finally`	Always runs (used for cleanup, like closing files)

32. Understanding multiple exceptions and custom exceptions.

Multiple `except` blocks	Handle different errors separately
Tuple in `except`	Catch multiple exceptions in one block
`raise`	Manually trigger an exception
Custom Exception Class	Create user-defined exceptions
Custom Exception with `__init__`	Add attributes to store extra error details

33.