**Que 1 : How functional programming works in Python.**

**What is Functional Programming?**

A way of writing code using **functions**, not changing data directly.

**Key Features:**

**1. Pure Functions**

Same input → same output

def add(x, y):

return x + y

**2. Lambda Function**

Short, one-line function

square = lambda x: x \* x

print(square(3)) # 9

**3. map()**

Applies a function to each item

nums = [1, 2, 3]

print(list(map(lambda x: x\*2, nums))) # [2, 4, 6]

**4. filter()**

Keeps items that match a condition

print(list(filter(lambda x: x % 2 == 0, nums))) # [2]

**5. reduce()**

Combines all items into one value

from functools import reduce

print(reduce(lambda x, y: x + y, nums)) # 6

**Que 2 : Using map(), reduce(), and filter() functions for processing data.**

**1. map() – Modify each element**

**Double each number:**

doubled = list(map(lambda x: x \* 2, numbers))

print(doubled) # [2, 4, 6, 8, 10, 12]

**2. filter() – Keep some elements**

**Keep only even numbers:**

evens = list(filter(lambda x: x % 2 == 0, numbers))

print(evens) # [2, 4, 6]

**3. reduce() – Combine all elements**

**Find the sum of all numbers:**

total = reduce(lambda x, y: x + y, numbers)

print(total) # 21

**Que 3 : Introduction to closures and decorators.**

**Closures**

A **closure** is a function **defined inside another function** that remembers the variables from the outer function even after it's finished executing.

Example:

def outer():

x = 10

def inner():

print(x) # remembers x even after outer() ends

return inner

func = outer()

func() # Output: 10

**Decorators**

A **decorator** is a function that **adds extra functionality** to another function **without changing its code**.

Example:

def my\_decorator(func):

def wrapper():

print("Before function call")

func()

print("After function call")

return wrapper

@my\_decorator

def greet():

print("Hello!")

greet()