**Module 14: Few more topics in-detailed**

* Filter

Takes a function and an iterable, returns only the items where the function evaluates to True.

* Signature: filter(function, iterable)
* Output: An iterator (often converted into list()).

Example

numbers = [10, 15, 20, 25, 30]

# keep only even numbers

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

print(list(evens)) # [10, 20, 30]

* 2. Map
* What it does: Applies a function to every item in an iterable.
* Signature: map(function, iterable)
* Output: An iterator (convert to list if needed).

Example

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

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

print(list(squares)) # [1, 4, 9, 16, 25]

* Reduce
* What it does: Applies a rolling computation to sequential pairs in an iterable.
* Imported from: from functools import reduce
* Signature: reduce(function, iterable)

Example1

from functools import reduce

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

# sum of all numbers

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

print(total) # 15

Example2: product of numbers

from functools import reduce

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

product = reduce(lambda x, y: x \* y, numbers)

print(product) # 120

* Decorators
* What they are: Functions that modify the behavior of another function without permanently changing it.
* Common use cases: Logging, authentication, caching, timing, etc.
* Syntax: @decorator\_name

Example1

def my\_decorator(func):

def wrapper():

print("Before the function runs")

func()

print("After the function runs")

return wrapper

@my\_decorator

def say\_hello():

print("Hello!")

say\_hello()

Example2:-Real Life

def log\_decorator(func):

def wrapper(\*args, \*\*kwargs):

print(f"Function {func.\_\_name\_\_} started...")

result = func(\*args, \*\*kwargs)

print(f"Function {func.\_\_name\_\_} finished.")

return result

return wrapper

@log\_decorator

def process\_payment(amount):

print(f"Processing payment of ₹{amount}...")

process\_payment(500)

* Frozen Set
* What it is: Like a normal set, but immutable (cannot be changed).
* Why useful: Can be used as a dictionary key or stored inside other sets.

normal\_set = set([1, 2, 3])

frozen = frozenset([1, 2, 3])

print(normal\_set)

print(frozen) # frozenset({1, 2, 3})

✅ Add element

normal\_set.add(4)

# ✅ Remove element

normal\_set.remove(2)

✅ Add element

frozen.add(4)

Note:-it will show error(frozen set is immutable)

Example:-

normal\_set = set([1, 2, 3])

frozen = frozenset([1, 2, 3])

# print(frozen)  # frozenset({1, 2, 3})

# normal\_set.add(4)

# normal\_set.remove(2)

frozen.aad(4)

print(normal\_set)

* Collections Module

Python’s collections provides specialized data structures beyond lists/tuples/dicts. Some key ones:

**a) Counter**

Counts hashable objects.

from collections import Counter

words = ["apple", "banana", "apple", "orange", "banana", "apple"]

print(Counter(words))

# Counter({'apple': 3, 'banana': 2, 'orange': 1})

**b) defaultdict**

from collections import defaultdict

employees = [

("John", "HR"),

("Alice", "IT"),

("Bob", "Finance"),

("Carol", "IT"),

("David", "HR"),

]

dept\_employees = defaultdict(list)

for name, dept in employees:

dept\_employees[dept].append(name) # if dept missing → defaultdict creates []

print(dept\_employees)

**c) namedtuple**

from collections import namedtuple

# Define a namedtuple for Employee

Employee = namedtuple("Employee", "name position salary")

# Create employee records

emp1 = Employee("Amit", "Manager", 85000)

emp2 = Employee("Nusrat", "Developer", 60000)

# Access fields by name

print(emp1.name, emp1.position, emp1.salary)

print(emp2.name, emp2.salary)

**d) deque**

Double-ended queue → fast appends/pops from both ends.

from collections import deque

dq = deque([1, 2, 3])

dq.appendleft(0)

dq.append(4)

print(dq) # deque([0, 1, 2, 3, 4])