**Session 1: Introduction to Data Structures**

**🔹 Types of Data Structures in Python**

| **Data Structure** | **Description** | **Advantages** | **Disadvantages** |
| --- | --- | --- | --- |
| **List** | Ordered, mutable collection | Easy to use, dynamic sizing | Slower for large data |
| **Tuple** | Ordered, immutable collection | Faster than lists, hashable | Cannot modify |
| **Set** | Unordered, unique elements | Fast membership testing | No indexing |
| **Dictionary** | Key-value pairs | Fast lookup, flexible | Keys must be unique |
| **Stack** | LIFO structure | Simple implementation | Limited access |
| **Queue** | FIFO structure | Useful in scheduling | Slower than deque |

**🔹 Implementing a Queue Using Two Stacks**

class QueueUsingStacks:

def \_\_init\_\_(self):

self.stack1 = []

self.stack2 = []

def enqueue(self, item):

self.stack1.append(item)

def dequeue(self):

if not self.stack2:

while self.stack1:

self.stack2.append(self.stack1.pop())

if self.stack2:

return self.stack2.pop()

else:

return "Queue is empty"

# Example usage

q = QueueUsingStacks()

q.enqueue(1)

q.enqueue(2)

print(q.dequeue()) # Output: 1

**🔹 Concepts with Examples**

**List:**

fr", "banana", "cherry"]

print(fruits[1]) # Output: banana

**Dictionary:**

person = {"name": "Alice", "age": 25}

print(person["name"]) # Output: Alice

**Queue (using collections.deque):**

from collections import deque

queue = deque()

queue.append("A")

queue.append("B")

print(queue.popleft()) # Output: A

**Stack (using list):**

stack = []

stack.append(10)

stack.append(20)

print(stack.pop()) # Output: 20

**Session 2 & 3: Functions and Decorators**

**🔹 Lambda Function**

A **lambda** is an anonymous function defined using the lambda keyword.

add = lambda x, y: x + y

print(add(2, 3)) # Output: 5

**Difference from regular function:**

* No name
* Single expression
* Used for short, throwaway functions

**🔹 Generator vs Normal Function**

**Normal Function:**

def get\_numbers():

return [1, 2, 3]

**Generator Function:**

def generate\_numbers():

for i in range(3):

yield i

gen = generate\_numbers()

print(next(gen)) # Output: 0

**🔹 Purpose of yield**

* yield pauses the function and saves its state.
* Used in **generators** to produce a sequence of values lazily.

def countdown(n):

while n &gt; 0:

yield n

n -= 1

**🔹 Decorators in Python**

Decorators modify the behavior of functions or classes.

def decorator\_func(func):

def wrapper():

print("Before function call")

func()

print("After function call")

return wrapper

@decorator\_func

def say\_hello():

print("Hello!")

say\_hello()

**🔹 Syntax of Applying a Decorator**

@decorator\_name

def function\_name():

pass

This is equivalent to:

function\_name = decorator\_name(function\_name)

**🔹 Role of @decorator Syntax**

* The @ symbol is syntactic sugar.
* It makes code cleaner and easier to read.
* Automatically applies the decorator to the function.