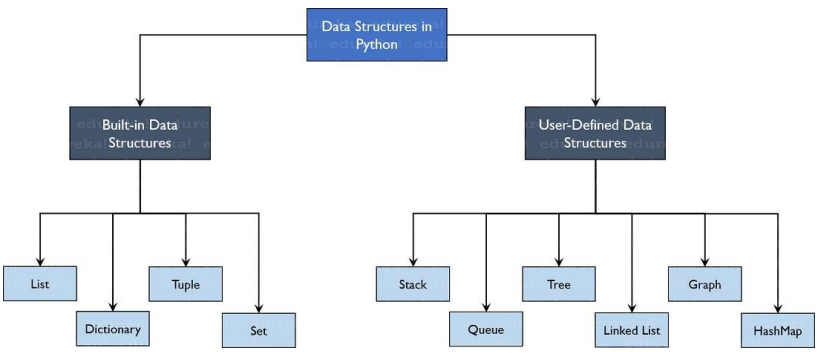
Data Structures in Python



User-Defined Data Structure

- **Stack:** Linear LIFO (Last-In-First-Out) Data structure
- **Queues:** Linear FIFO (First-In-First-Out) data structure
- **Linked Lists:** Linear data structures that are linked with pointers
- **Trees:** Non-Linear data structures having a root and nodes
- **Graphs:** Store a collection of points or nodes along with edges
- **Hash Maps:** In Python, Hash Maps are the same as Dictionaries

Stack

A stack is a linear data structure that stores items in a Last-In/First-Out (LIFO) or First-In/Last-Out (FILO) manner. In stack, a new element is added at one end and an element is removed from that end only. The insert and delete operations are usually called push and pop.

Python's built-in data structure list can be used as a stack. Instead of push(), append() is used to add elements to the top of stack while pop() removes the element in LIFO order.

Implementation of Stack using List

In [ ]:
```
MyStack = list()

MyStack.append("Laiqa Imran")
MyStack.append("Lab Instructor")
MyStack.append("FOIT")

display(MyStack)

MyStack.pop()
```

Queue

Like stack, queue is a linear data structure that stores items in First In First Out (FIFO) manner. With a queue the least recently added item is removed first.

Python's built-in data structure List can be used as a queue. Instead of enqueue() and dequeue(), append() and pop() function is used.

Implementation of Queue using List

In [ ]:
```
MyQueue = list()

MyQueue.append("Laiqa Imran")
MyQueue.append("Lab Instructor")
MyQueue.append("FOIT")

display(MyQueue)

MyQueue.pop(0)
```

Trees

Tree represents the nodes connected by edges. It is a non-linear data structure. It has the following properties.

- One node is marked as Root node.
- Every node other than the root is associated with one parent node.
- Each node can have an arbiatry number of chid node.

In [ ]:
```
class Node:
    def __init__(self, Value):
        self.Left = None
        self.Data = Value
        self.Right = None
class Tree:
    def CreateNode(self, Data):
        return Node(Data)
    def Insert(self, Node , Data):
        #if tree is empty , return a root node
        if Node is None:
            return self.CreateNode(Data)
        # if data is smaller than parent , insert it into left side
        if Data < Node.Data:
            Node.Left = self.Insert(Node.Left, Data)
        elif Data > Node.Data:
            Node.Right = self.Insert(Node.Right, Data)
        return Node
    def Treeprint(self, Root):
        if Root is not None:
            self.Treeprint(Root.Left)
            print(Root.Data)
            self.Treeprint(Root.Right)
Root = None
MyTree = Tree()
Root = MyTree.Insert(Root, 10)
MyTree.Insert(Root, 20)
MyTree.Insert(Root,  30)
MyTree.Insert(Root,  40)
MyTree.Insert(Root,  70)
MyTree.Insert(Root,  60)
MyTree.Insert(Root, 80)
MyTree.Treeprint(Root)
```

In [ ]:
```
from anytree import Node, RenderTree

CC = Node("Imran Arshad")
Teacher1 = Node("Mohsin Abbas", parent=CC)
Teacher2 = Node("Freeha Iqbal", parent=CC)
Teacher3 = Node("Madiha Yousaf", parent=CC)
LI = Node("Samman Ashraf", parent=Teacher1)
LI = Node("Abdul Mateen", parent=Teacher1)
LI = Node("Samman Ashraf", parent=Teacher2)
LI = Node("Abdul Mateen", parent=Teacher2)
LI = Node("Samman Ashraf", parent=Teacher3)
LI = Node("Abdul Mateen", parent=Teacher3)

for pre, fill, node in RenderTree(CC):
    print("%s%s" % (pre, node.name))
```