Lists

Python Lists are ordered collections of data just like arrays in other programming languages. It allows different types of elements in the list. The implementation of Python List is similar to Vectors in C++ or ArrayList in JAVA. The costly operation is inserting or deleting the element from the beginning of the List as all the elements are needed to be shifted. Insertion and deletion at the end of the list can also become costly in the case where the preallocated memory becomes full.

Example: Creating Python List

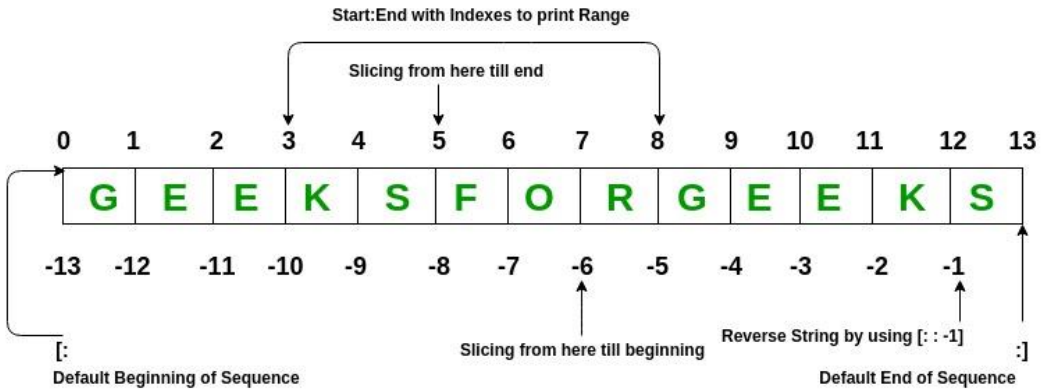
Python3

```
List = [1, 2, 3, "GFG", 2.3]
print(List)
```

Output

```
[1, 2, 3, 'GFG', 2.3]
```

List elements can be accessed by the assigned index. In python starting index of the list, a sequence is 0 and the ending index is (if N elements are there) N-1.



Example: Python List Operations

Python3

```
# Creating a List with
# the use of multiple values
List = ["Geeks", "For", "Geeks"]
print("\nList containing multiple values: ")
print(List)

# Creating a Multi-Dimensional List
# (By Nesting a list inside a List)
List2 = [['Geeks', 'For'], ['Geeks']]
print("\nMulti-Dimensional List: ")
print(List2)

# accessing a element from the
# list using index number
print("Accessing element from the list")
print(List[0])
print(List[2])

# accessing a element using
# negative indexing
print("Accessing element using negative indexing")

# print the last element of list
print(List[-1])

# print the third last element of list
print(List[-3])
```

Output

```
List containing multiple values:
['Geeks', 'For', 'Geeks']


Multi-Dimensional List:
[['Geeks', 'For'], ['Geeks']]
Accessing element from the list
Geeks
Geeks
Accessing element using negative indexing
Geeks
Geeks
```

Tuple

Python tuples are similar to lists but Tuples are immutable in nature i.e. once created it cannot be modified. Just like a List, a Tuple can also contain elements of various types.

In Python, tuples are created by placing a sequence of values separated by 'comma' with or without the use of parentheses for grouping of the data sequence.

Note: To create a tuple of one element there must be a trailing comma. For example, (8,) will create a tuple containing 8 as the element.

Example: Python Tuple Operations

Python3

```
# Creating a Tuple with
# the use of Strings
Tuple = ('Geeks', 'For')
print("\nTuple with the use of String: ")
print(Tuple)

# Creating a Tuple with
# the use of list
list1 = [1, 2, 4, 5, 6]
print("\nTuple using List: ")
Tuple = tuple(list1)

# Accessing element using indexing
print("First element of tuple")
print(Tuple[0])

# Accessing element from last
# negative indexing
print("\nLast element of tuple")
print(Tuple[-1])

print("\nThird last element of tuple")
print(Tuple[-3])
```

Output

```
Tuple with the use of String:

('Geeks', 'For')


Tuple using List:

First element of tuple

1


Last element of tuple

6


Third last element of tuple
```

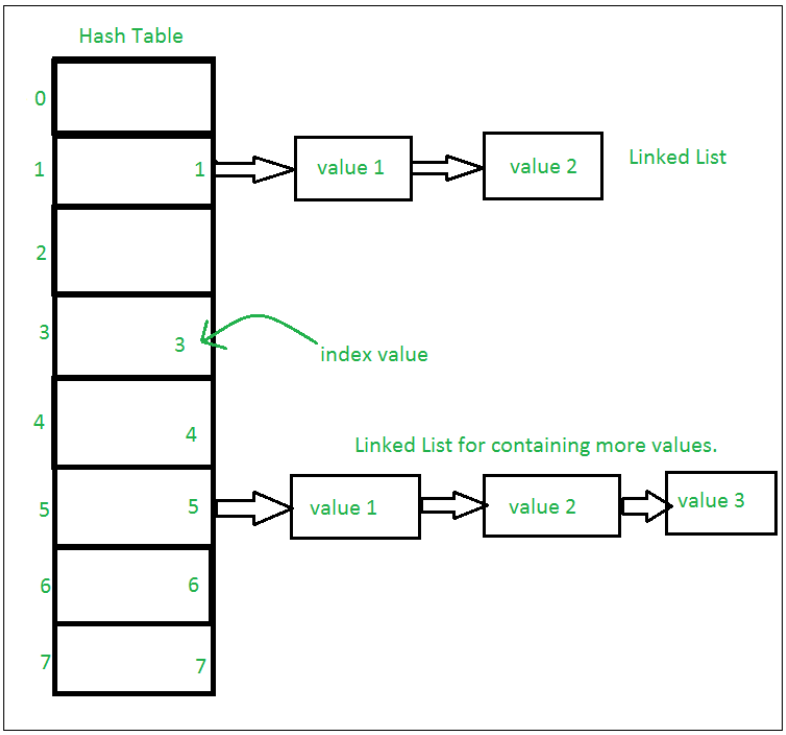
4

Set

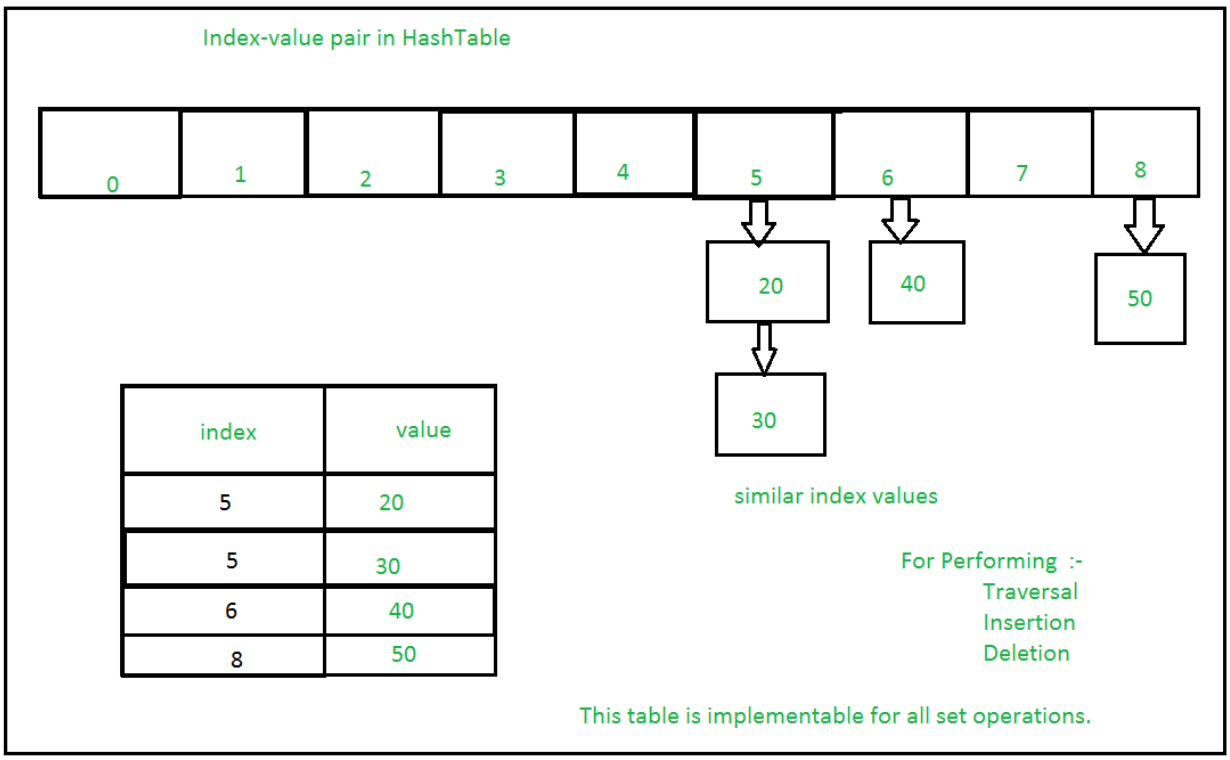
Python set is a mutable collection of data that does not allow any duplication. Sets are basically used to include membership testing and eliminating duplicate entries. The data structure used in this is Hashing, a popular technique to perform insertion, deletion, and traversal in O(1) on average.

If Multiple values are present at the same index position, then the value is appended to that index position, to form a Linked List. In, CPython Sets are implemented using a dictionary with dummy variables, where key beings the members set with greater optimizations to the time complexity.

Set Implementation:



Sets with Numerous operations on a single HashTable:



Example: Python Set Operations

Python3

```
# Creating a Set with
# a mixed type of values
# (Having numbers and strings)
Set = set([1, 2, 'Geeks', 4, 'For', 6, 'Geeks'])
print("\nSet with the use of Mixed Values")
print(Set)

# Accessing element using
# for loop
print("\nElements of set: ")
for i in Set:
    print(i, end =" ")
print()

# Checking the element
# using in keyword
print("Geeks" in Set)
```

Output

```
Set with the use of Mixed Values

{1, 2, 4, 6, 'For', 'Geeks'}


Elements of set:

1 2 4 6 For Geeks
```

```
True
```

Frozen Sets

Frozen sets in Python are immutable objects that only support methods and operators that produce a result without affecting the frozen set or sets to which they are applied. While elements of a set can be modified at any time, elements of the frozen set remain the same after creation.

Example: Python Frozen set

Python3

```
# Same as {"a", "b","c"}
normal_set = set(["a", "b","c"])

print("Normal Set")
print(normal_set)

# A frozen set
frozen_set = frozenset(["e", "f", "g"])

print("\nFrozen Set")
print(frozen_set)

# Uncommenting below line would cause error as
# we are trying to add element to a frozen set
# frozen_set.add("h")
```

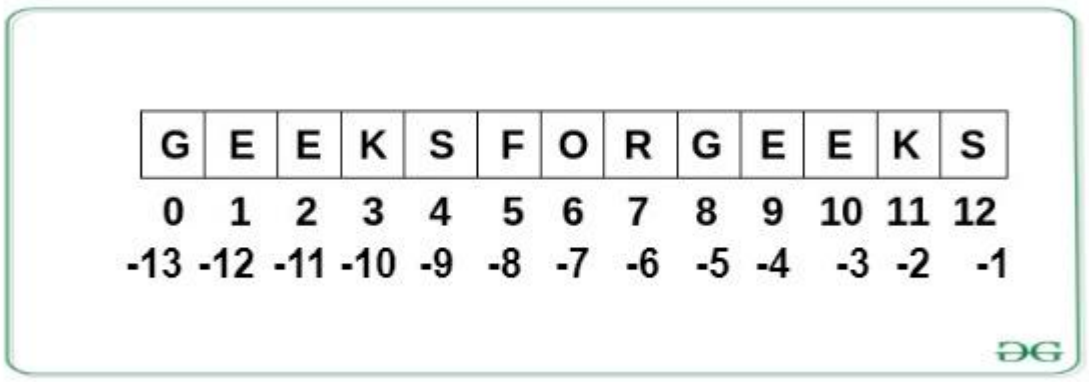
Output

```
Normal Set
{'a', 'b', 'c'}


Frozen Set
frozenset({'f', 'g', 'e'})
```

String

Python Strings is the immutable array of bytes representing Unicode characters. Python does not have a character data type, a single character is simply a string with a length of 1.

Note: As strings are immutable, modifying a string will result in creating a new copy.



Example: Python Strings Operations

Python3

```
String = "Welcome to GeeksForGeeks"
print("Creating String: ")
print(String)

# Printing First character
print("\nFirst character of String is: ")
print(String[0])

# Printing Last character
print("\nLast character of String is: ")
print(String[-1])
```

Output

```
Creating String:
Welcome to GeeksForGeeks


First character of String is:
W


Last character of String is:
s
```

Dictionary

Python dictionary is an unordered collection of data that stores data in the format of key:value pair. It is like hash tables in any other language with the time complexity of O(1). Indexing of Python Dictionary is done with the help of keys. These are of any hashable type i.e. an object whose can never change like strings, numbers, tuples, etc. We can create a dictionary by using curly braces ({}) or dictionary comprehension.

Example: Python Dictionary Operations

Python3

```
# Creating a Dictionary
Dict = {'Name': 'Geeks', 1: [1, 2, 3, 4]}
print("Creating Dictionary: ")
print(Dict)

# accessing a element using key
print("Accessing a element using key:")
print(Dict['Name'])

# accessing a element using get()
# method
print("Accessing a element using get:")
print(Dict.get(1))

# creation using Dictionary comprehension
myDict = {x: x**2 for x in [1,2,3,4,5]}
print(myDict)
```

Output

```
Creating Dictionary:

{'Name': 'Geeks', 1: [1, 2, 3, 4]}

Accessing a element using key:

Geeks

Accessing a element using get:

[1, 2, 3, 4]

{1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
```

Matrix

A matrix is a 2D array where each element is of strictly the same size. To create a matrix we will be using the NumPy package.

Example: Python NumPy Matrix Operations

Python3

```
import numpy as np

a = np.array([[1,2,3,4],[4,55,1,2],
              [8,3,20,19],[11,2,22,21]])
m = np.reshape(a,(4, 4))
print(m)

# Accessing element
print("\nAccessing Elements")
```

```
print(a[1])
print(a[2][0])

# Adding Element
m = np.append(m,[[1, 15,13,11]],0)
print("\nAdding Element")
print(m)

# Deleting Element
m = np.delete(m,[1],0)
print("\nDeleting Element")
print(m)
```

Output

```
[[ 1  2  3  4]
 [ 4 55  1  2]
 [ 8  3 20 19]
 [11  2 22 21]]

Accessing Elements
[ 4 55  1  2]
8

Adding Element
[[ 1  2  3  4]
 [ 4 55  1  2]
 [ 8  3 20 19]
 [11  2 22 21]
 [ 1 15 13 11]]

Deleting Element
[[ 1  2  3  4]
 [ 8  3 20 19]
 [11  2 22 21]
 [ 1 15 13 11]]
```

Bytearray

Python Bytearray gives a mutable sequence of integers in the range 0 <= x < 256.

Example: Python Bytearray Operations

Python3

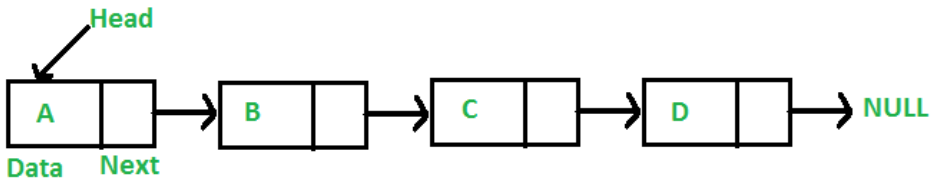
```
# Creating bytearray
a = bytearray((12, 8, 25, 2))
print("Creating Bytearray:")
print(a)

# accessing elements
print("\nAccessing Elements:", a[1])

# modifying elements
a[1] = 3
print("\nAfter Modifying:")
print(a)

# Appending elements
a.append(30)
print("\nAfter Adding Elements:")
print(a)
```

Output

```
Creating Bytearray:
bytearray(b'\x0c\x08\x19\x02')

Accessing Elements: 8

After Modifying:
bytearray(b'\x0c\x03\x19\x02')

After Adding Elements:
bytearray(b'\x0c\x03\x19\x02\x1e')
```

Linked List

A linked list is a linear data structure, in which the elements are not stored at contiguous memory locations. The elements in a linked list are linked using pointers as shown in the below image:



A linked list is represented by a pointer to the first node of the linked list. The first node is called the head. If the linked list is empty, then the value of the head is NULL. Each node in a list consists of at least two parts:

- Data
- Pointer (Or Reference) to the next node

Example: Defining Linked List in Python

Python3

```
# Node class
class Node:

    # Function to initialize the node object
    def __init__(self, data):
        self.data = data # Assign data
        self.next = None # Initialize
                         # next as null

# Linked List class
class LinkedList:

    # Function to initialize the Linked
    # List object
    def __init__(self):
        self.head = None
```

Let us create a simple linked list with 3 nodes.

Python3

```
# A simple Python program to introduce a linked list

# Node class
class Node:

    # Function to initialise the node object
    def __init__(self, data):
        self.data = data # Assign data
        self.next = None # Initialize next as null


# Linked List class contains a Node object
class LinkedList:

    # Function to initialize head
    def __init__(self):
        self.head = None


# Code execution starts here
if __name__=='__main__':
```

```
# Start with the empty list
llist = LinkedList()

llist.head = Node(1)
second = Node(2)
third = Node(3)

'''
Three nodes have been created.
We have references to these three blocks as head,
second and third

llist.head     second             third
    |             |                |
    |             |                |
+----+------+     +----+------+     +----+------+
| 1  | None |     | 2  | None |     | 3  | None |
+----+------+     +----+------+     +----+------+
'''

llist.head.next = second; # Link first node with second

'''
Now next of first Node refers to second. So they
both are linked.

llist.head     second             third
    |             |                |
    |             |                |
+----+------+     +----+------+     +----+------+
| 1  | o-------->| 2  | null |     | 3  | null |
+----+------+     +----+------+     +----+------+
'''

second.next = third; # Link second node with the third node

'''
Now next of second Node refers to third. So all three
nodes are linked.

llist.head     second             third
    |             |                |
    |             |                |
+----+------+     +----+------+     +----+------+
| 1  | o-------->| 2  | o-------->| 3  | null |
+----+------+     +----+------+     +----+------+
'''
```

Linked List Traversal

In the previous program, we have created a simple linked list with three nodes. Let us traverse the created list and print the data of each node. For traversal, let us write a general-purpose function printList() that prints any given list.

Python3

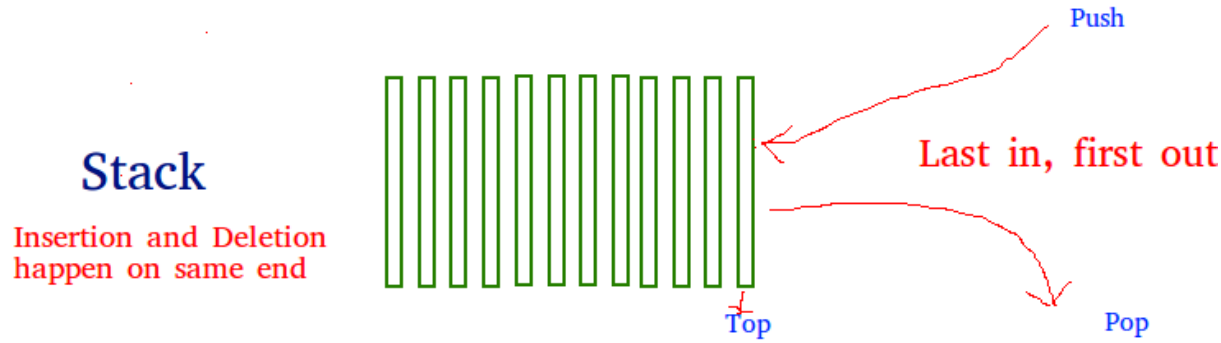
```
# A simple Python program for traversal of a linked list

# Node class
class Node:

    # Function to initialise the node object
    def __init__(self, data):
        self.data = data # Assign data
        self.next = None # Initialize next as null


# Linked List class contains a Node object
class LinkedList:

    # Function to initialize head
    def __init__(self):
        self.head = None

    # This function prints contents of linked list
    # starting from head
    def printList(self):
        temp = self.head
        while (temp):
            print (temp.data)
            temp = temp.next


# Code execution starts here
if __name__=='__main__':

    # Start with the empty list
    llist = LinkedList()

    llist.head = Node(1)
    second = Node(2)
    third = Node(3)

    llist.head.next = second; # Link first node with second
    second.next = third; # Link second node with the third node

    llist.printList()
```

Output

```
1

2

3
```

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 often called push and pop.



The functions associated with stack are:

- **empty() –** Returns whether the stack is empty – Time Complexity: O(1)
- **size() –** Returns the size of the stack – Time Complexity: O(1)
- **top() –** Returns a reference to the topmost element of the stack – Time Complexity: O(1)
- **push(a) –** Inserts the element 'a' at the top of the stack – Time Complexity: O(1)
- **pop() –** Deletes the topmost element of the stack – Time Complexity: O(1)

Python3

```
stack = []

# append() function to push
# element in the stack
stack.append('g')
stack.append('f')
stack.append('g')

print('Initial stack')
print(stack)

# pop() function to pop
# element from stack in
# LIFO order
print('\nElements popped from stack:')
print(stack.pop())
print(stack.pop())
print(stack.pop())

print('\nStack after elements are popped:')
print(stack)

# uncommenting print(stack.pop())
# will cause an IndexError
# as the stack is now empty
```

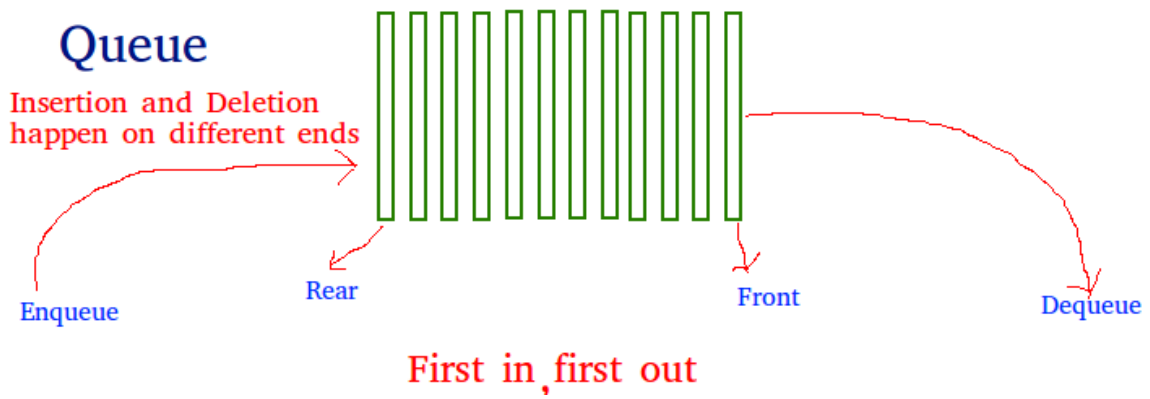
Output

```
Initial stack

['g', 'f', 'g']
```

```
Elements popped from stack:
g
f
g

Stack after elements are popped:
[]
```

Queue

As a stack, the queue is a linear data structure that stores items in a First In First Out (FIFO) manner. With a queue, the least recently added item is removed first. A good example of the queue is any queue of consumers for a resource where the consumer that came first is served first.



Operations associated with queue are:

- **Enqueue:** Adds an item to the queue. If the queue is full, then it is said to be an Overflow condition – Time Complexity: O(1)
- **Dequeue:** Removes an item from the queue. The items are popped in the same order in which they are pushed. If the queue is empty, then it is said to be an Underflow condition – Time Complexity: O(1)
- **Front:** Get the front item from queue – Time Complexity: O(1)
- **Rear:** Get the last item from queue – Time Complexity: O(1)

Python3

```
# Initializing a queue
queue = []

# Adding elements to the queue
queue.append('g')
queue.append('f')
queue.append('g')
```

```
print("Initial queue")
print(queue)

# Removing elements from the queue
print("\nElements dequeued from queue")
print(queue.pop(0))
print(queue.pop(0))
print(queue.pop(0))

print("\nQueue after removing elements")
print(queue)

# Uncommenting print(queue.pop(0))
# will raise and IndexError
# as the queue is now empty
```

Output

```
Initial queue
['g', 'f', 'g']


Elements dequeued from queue
g
f
g


Queue after removing elements
[]
```