UNIT 1

BASIC CONCEPTS OF DATA STRUCTURES

1.1 Data Structure Basic Concepts :

Data Structure (Definition)

- Data Structure is a mathematical or logical way of organizing data in memory. Data Structure does not represent only data in memory but it also represents the relationship among the data that is stored in memory.
- There are various operations that can be performed on Data Structure :
  (1) Traversal (4) Searching
  (2) Insertion (5) Sorting
  (3) Deletion (6) Merging
- Data Structure allows us to manipulate data by specifying a set of values, set of operations that can be performed on set of values and set of rules that needs to be followed while performing operations.

1.2 Types of data structures :

- There are two types of Data Structures :
  - (1) Linear Data Structure
  - (2) Non Linear Data Structure

1.2.1 Linear Data Structure :

- The Data Structure in which elements are arranged such that we can process them in linear fashion (sequentially) is called linear data structure.
- Examples of Linear Data Structures are :

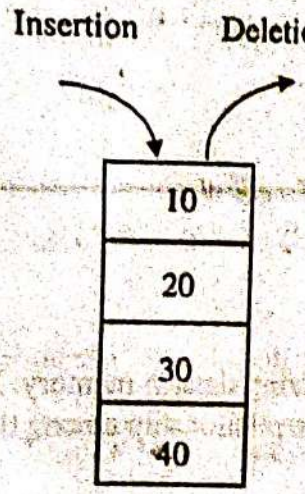
(1) Array :

- Array is a collection of variables of same data type that share common name.
- Array is an ordered set which consist of fixed number of elements.
- In array memory is allocated sequentially to each element. So it is also known as sequential list.

2000	10
2002	20
2004	30
2006	40
2008	50

(2) Stack :

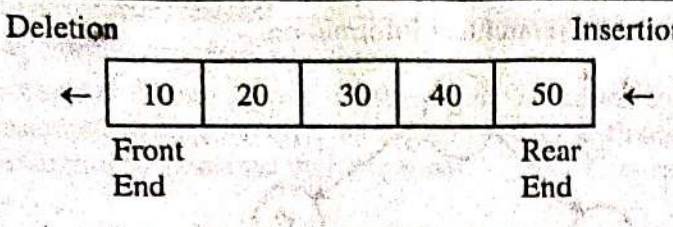
- Stack is a linear Data Structure in which insertion and deletion operation are performed at same end.
- Following figure represents Stack :



- Since insertion and deletion operations are performed at same end, the element which is inserted Last is first to delete. So Stack is also known as **last in First out (LIFO)**.

(3) Queue :

- Queue is a linear Data Structure in which insertion operation is performed at one end called Rear end and deletion operation is performed at another end called front end.
- Following figure represents Queue :

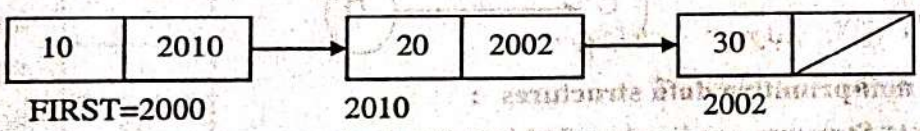


- Since insertion operation is performed at rear end and deletion operation is performed at front end the element which is inserted first is first to delete. So Queue is also known as **First in First out (FIFO).**
- Following are four Types of Queue :
  (1) Simple Queue
  (2) Circular Queue
  (3) Double ended Queue
  (4) Priority Queue

(4) Linked List :

- Linked list is an ordered set which consist of variable number of elements.
- In Linked list elements are logically adjacent to each other but they are physically not adjacent. It means elements of linked list are not sequentially stored in memory.
- Linked list is a collection of nodes. Each node consists of two parts. First part represents value of the node and second part represents an address of the next node. If Next node is not present then it contains NULL value.

- Consider Following Presentation of Linked List :



- There are four types of Linked List :
  (1) Singly Linked List
  (2) Doubly Linked List
  (3) Circular Singly Linked List
  (4) Circular Doubly Linked List

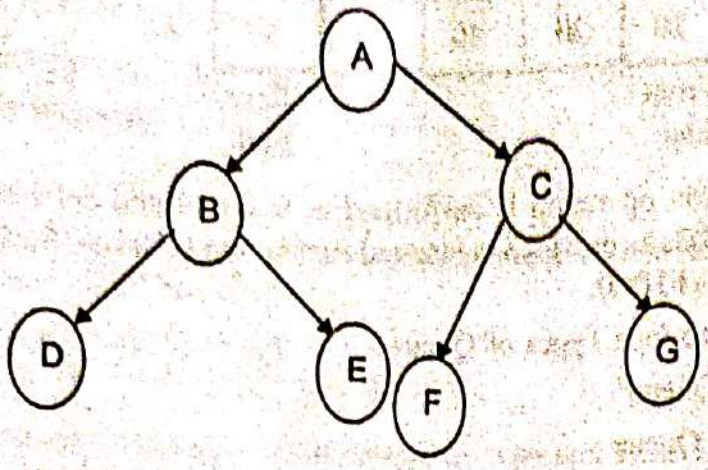
1.2.2 Non Linear Data Structure :

- The Data Structure in which elements are arranged such that we can not process them in linear fashion (sequentially) is called Non-Linear data structure.
- Non Linear Data Structures are useful to represent more complex relationships as compared to Linear Data Structures.
- Examples of Non-Linear Data Structures are :

(1) Tree :

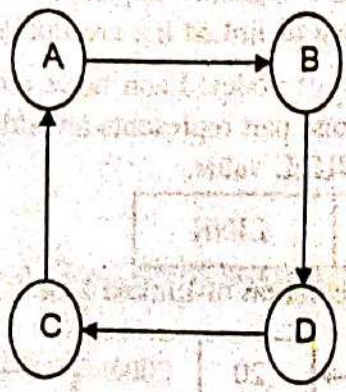
- A tree is defined as a finite set of one or more nodes such that :
  (1) There is a special node called root node having no predecessor.
  (2) All the nodes in a tree except root node having only one predecessor.
  (3) All the nodes in a tree having or more successor.

- A tree is used to represent hierarchical information.



(2) Graph :

- A Graph is a collection of nodes and edges which is used to represent relationship between pair of nodes.
- A graph G consists of set of nodes and set of edges and a mapping from the set of edges to a set of pairs of nodes.
- Following figure represents Graph.



1.2.3 Primitive and non-primitive data structures :

- Generally Data Structure can be classified into two basic categories:
  (1) Primitive Data Structure
  (2) Non Primitive Data Structure
- **Primitive Data Structure:**
- The Data Structure which is directly operated by machine level instruction is known as Primitive Data Structure.
- All primary (built in) data types are known as Primitive data Structure.
- Following are Primitive Data Structure :

(1) Integer :

- Integer Data Structure is used to represent numbers without decimal points.
- For Example: 23, -45, 11 etc...
- Using this Data Structure we can represent positive as well as negative numbers without decimal point.

(2) Float :

- Float Data Structure is used to represent numbers having decimal point.

- For Example: 3.2, 4.56 etc....
- In computer floating point numbers can be represented using normalized floating point representation. In this type of representation a floating point number is expressed as a combination of mantissa and exponent.

(3) Character :

- Character is used to represent single character enclosed between single inverted comma.
- It can store letters (a-z, A-Z), digit (0-9) and special symbols.

- **Non Primitive Data Structure :**
  - The Data Structure which is not directly operated by machine level instruction is known as Non Primitive Data Structure.
  - Non Primitive Data Structure is derived from Primitive Data Structure.
  - Non Primitive Data Structure are classified into two categories :
- **Linear Data Structure :**
  - The Data Structure in which elements are arranged such that we can process them in linear fashion (sequentially) is called linear data structure.
  - Examples of Linear Data Structures are :

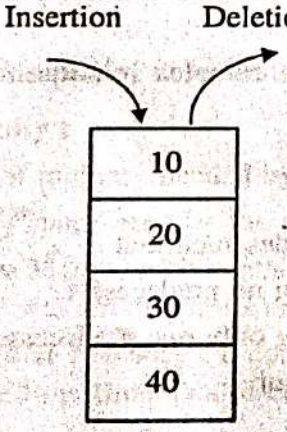
(1) Array :

- Array is a collection of variables of same data type that share common name.
- Array is an ordered set which consist of fixed number of elements.
- In array memory is allocated sequentially to each element. So it is also known as sequential list.

2000	10
2002	20
2004	30
2006	40
2008	50

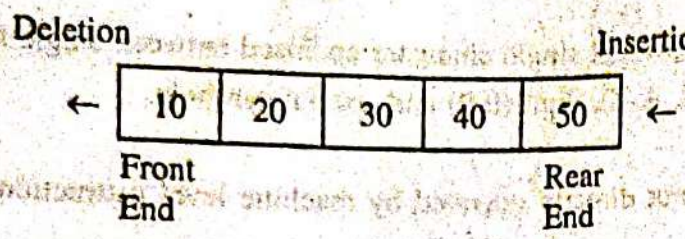
(2) Stack :

- Stack is a linear Data Structure in which insertion and deletion operation are performed at same end.
- Following figure represents Stack :



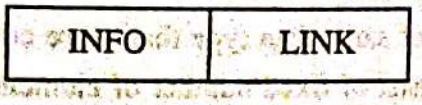
(3) Queue :

- Queue is a linear Data Structure in which insertion operation is performed at one end call Rear end and deletion operation is performed at another end called front end.
- Following figure represents Queue :

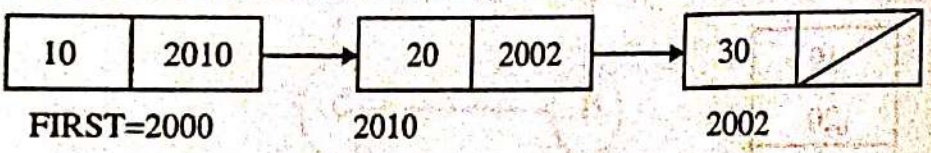


(4) Linked List :

- Linked list is an ordered set which consist of variable number of elements.
- In Linked list elements are logically adjacent to each other but they are physically no adjacent. It means elements of linked list are not sequentially stored in memory.
- Linked list is a collection of nodes. Each node consists of two parts. First part represent value of the node and second part represents an address of the next node. If Next node is no present then it contains NULL value.



- Consider Following Presentation of Linked List :



- There are four types of Linked List :
  (1) Singly Linked List
  (2) Doubly Linked List
  (3) Circular Singly Linked List
  (4) Circular Doubly Linked List

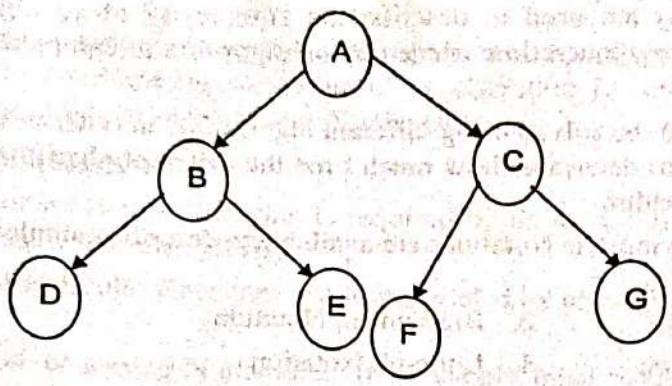
Non Linear Data Structures :

- The Data Structure in which elements are arranged such that we can not process them in linear fashion (sequentially) is called Non-Linear data structure.
- Non Linear Data Structures are useful to represent more complex relationships as compared to Linear Data Structures.
- Examples of Non-Linear Data Structures are :

(1) Tree

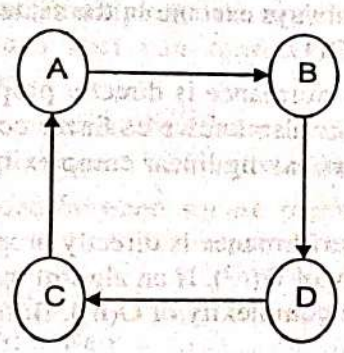
- A tree is defined as a finite set of one or more nodes such that :
  (1) There is a special node called root node having no predecessor.
  (2) All the nodes in a tree except root node having only one predecessor.
  (3) All the nodes in a tree having or more successor.

- A tree is used to represent hierarchical information.



(2) Graph :

- A Graph is a collection of nodes and edges which is used to represent relationship between pair of nodes.
- A graph G consists of set of nodes and set of edges and a mapping from the set of edges to a set of pairs of nodes.
- Following figure represents Graph.



1.3 Analysis Terms (for the definitions purpose only) :

1.3.1 Time Complexity :

- The amount of time needed by a program to complete its execution is known as Time complexity.
- The measurement of time is done in terms of number of instructions executed by the program during its execution.
- Thus Time Complexity depends on the Size of the program and type of the algorithm being used.

1.3.2 Space Complexity :

- The amount of memory needed by a program during its execution is known as Space complexity.
- Total memory space need by the program is the sum of following two memory :

(1) Fixed size Memory : It contains the space required for simple variables, constants, instructions and fixed size structured variable such as array.

(2) Variable size Memory : It contains the space required for structured variable to which memory is allocated run time. It also contains space required while function is calling itself.

1.3.3 Asymptotic Notations :

- Asymptotic Notations are used to describe the complexity of an algorithm. Complexity of an algorithm indicates how much time needed by an algorithm to complete its execution for given set of input data.
- The same problem can be solved using different algorithms. In order to select the best algorithm for a problem, we need to determine how much time the different algorithm will take to run and then select the better algorithm.
- There are various Asymptotic Notations are available to describe complexity of an algorithm. Which are
  1. Big-Oh Notation
  2. Big-Omega Notation
  3. Big-Theta Notation
  4. Little-oh Notation
  5. Little-omega Notation

1.3.4 Big 'O' Notation :

- Big - O Notation is used to describe the Time complexity of an algorithm. It means how much time is needed by an algorithm to complete its execution for the input size of N. For Example a sorting algorithm take longer time to sort 5000 elements than 50.
- Following are commonly used Orders of an algorithm.
  - **(1) O(1) :** An algorithm that will always execute in the same time regardless of the size of the input data is having complexity of O(1).
  - **(2) O(n) :** An algorithm whose performance is directly proportional to the size of the input data is having complexity of O(n). It is also known as linear complexity. If an algorithm uses looping structure over the data then it is having linear complexity of O(n). Linear Search is an example of O(n).
  - **(3) $O(n^2)$:** An algorithm whose performance is directly proportional to the square of the size of the input data is having complexity of $O(n^2)$. If an algorithms uses nested looping structure over the data then it is having quadratic complexity of $O(n^2)$. Bubble sort, Selection Sort are the example of $O(n^2)$.
  - **(4) O(logn) :** An algorithm in which during each iteration the input data set is partitioned into to sub parts is having complexity of O(logn). Quick Sort, Binary Search are the example of O(logn) complexity.

1.3.5 Best case Time Complexity :

- The measurement of minimum time that is required by an algorithm to complete its execution is known as Best Case Time Complexity.
- Time complexity of particular algorithm can be calculated by providing different input values to the algorithm.
- Consider an example of sorting N elements. If we supply input values that is already sorted, an algorithm required less time to sort them. This is known as Best case time complexity.
- However best case time complexity does not guarantee that the algorithm will always execute within this time for different input values.

1.3.6 Average case Time Complexity :

- The measurement of average time that is required by an algorithm to complete its execution is known as Average Case Time Complexity.

- Time complexity of particular algorithm can be calculated by providing different input values to the algorithm.
- Consider an example of sorting N elements. Average time complexity can be calculated by measuring the time required to complete the execution of an algorithm for different input values and then calculate the average time required to sort N elements.

1.3.7 Worst case Time Complexity :

- The measurement of maximum time that is required by an algorithm to complete its execution is known as Worst Case Time Complexity.
- Time complexity of particular algorithm can be calculated by providing different input values to the algorithm.
- Consider an example of sorting N elements. If we supply input values that is in reverse order, an algorithm required maximum time to sort them. This is known as worst case time complexity.
- Thus, worst case time complexity always guarantees that the algorithm will always execute within this time for different input values.

1.4 Python Specific Data Structures :

1.4.1 Lists and operations on Lists :

- List is an **ordered set** which consist of variable number of elements.
- In Python List is used to represent multiple values under single variable. Elements of the List need not be of same type. It means List can represents homogeneous(same) as well as non homogeneous(different) elements.
- List is ordered means all the elements in the List are assigned index starting from 0. Index of first element in the list is 0, second element is 1 and so on.
- Elements of the List are enclosed between square parenthesis [ ] and separated by comma.
- **Example :**

```
MyList = ["Red", "Green", "Blue"]
print(MyList[0])          #Print Red
```

- List is **changeable (mutable).** Once List is created we can add new elements in it as well as update value of particular element in the List.

List Operations :

- Various operations that can be performed on list are :

(1) Creating a List
(2) Access individual Element of the List
(3) Traversing in a List
(4) Searching a List
(5) List Concatenation
(6) List Repetition

(1) Creating List :

- General Syntax for creating list in Python is :

```
List_Name = [Element1,Element2,...,Element-N]
```

- Consider Following example which creates a list named fruit and display its value :

```
fruit = ["Apple","Banana","WaterMelon","Grapes"]
print(fruit)
```

- List can also be created using list() constructor.

❖ Example :

```
fruit = list(("Apple","Banana","WaterMelon","Grapes"))
print(fruit)
```

(2) Access Element in List :

❖ Individual element of the list can be accessed by using its index position along with name of Li In list index start from 0.

❖ Example :

```
fruit = ["Apple","Banana","WaterMelon","Grapes"]
print(fruit[1]) # Print Banana
```

❖ Note : If specified index is not present in the list than it will give index out of range error.

❖ Index can be positive or negative. If positive index is specified than it will access element fro left side and if negative index is specified then it will access element from right side.

❖ Example :

```
fruit = ["Apple","Banana","WaterMelon","Grapes"]
print(fruit[0]) # Print Apple
print(fruit[-2]) # Print WaterMelon
```

(3) Access Range of Elements in List :

❖ You can access range of elements from list by specifying start index and stop index separated colon(:).

❖ Example :

```
fruit = ["Apple", "Banana", "WaterMelon", "Grapes"]
print(fruit[1:3])      # Print ['Banana' , 'WaterMelon']
```

❖ Note : Element at Start Index is included but Element at stop index is excluded.

You can specify negative index also to access range of elements from List. Where -1 indica index of last element,-2 indicates index of second last element and so on.

❖ Example :

```
fruit = ["Apple","Banana","WaterMelom","Grapes"]
print(fruit[-4:-2]) # Print ['Apple', 'Banana']
```

❖ If you don't specify start index than by default start index is considered as 0.

```
fruit = ["Apple","Banana","WaterMelom","Grapes"]
print(fruit[ :2])   # Print ['Apple', 'Banana']
```

❖ If you don't specify stop index than by default stop index is considered as index of last eleme in the List.

```
fruit = ["Apple","Banana","WaterMelom","Grapes"]
print(fruit[1:]) # Print ['Banana','WaterMelon','Grapes']
```

(4) Traversing a List :

❖ Traversing a list means access each element of list. In Python we can traverse a list using for loo or while loop.

❖ Example : (Using for loop)

```
fruit = ["Apple","Banana","WaterMelom","Grapes"]
for x in fruit:
    print(x)
```

❖ **Example : (Using while loop)**

```
fruit = ["Apple","Banana","WaterMelom","Grapes"]
x = 0
while(x < len(fruit)):
    print(fruit[x])
    x = x +
```

(5) Searching a List :

❖ You can search elements in a list using Membership operator.

❖ Membership Operators(in not in) are used to check weather particular element exist in the List or not.

❖ **Example :**

```
fruit = ["Apple","Banana","WaterMelom","Grapes"]
print("Mango" in fruit)          # False
print("Banana" in fruit) # True
```

(6) Concatenating List :

❖ Multiple list can be concatenate using + operator in python.

❖ **Example :**

```
fruit = ["Apple","Banana","WaterMelom","Grapes"]
vegetable = ["Potato","Tomato","Brinjal"]
Mix = fruit+vegetable
print(Mix)
```

❖ It will display

['Apple', 'Banana', 'WaterMelom', 'Grapes', 'Potato', 'Tomato', 'Brinjal']

(7) Repeating List :

❖ Repeated list can be created using * operator.

❖ **Example :**

```
fruit = ["Apple","Banana","WaterMelom","Grapes"]
print(fruit*2)
```

❖ It will display

['Apple', 'Banana', 'WaterMelom', 'Grapes', 'Apple', 'Banana', 'WaterMelom', 'Grapes']

List Methods and Built-in Functions :

❖ List has several built in methods to perform various operations on it.

Method	Purpose

append()	✧ Add new element at the end of List.
	✧ **Example :**
	`List1 = [1, 2, 3, 4, 5]`
	`print(List1)`
	`List1.append(6)`
	`print(List1)`

	✧ **Output :** [1, 2, 3, 4, 5] [1, 2, 3, 4, 5, 6]

insert()	✧ Add new element at specific index in the List.
	✧ **Example :**
	`List1 = [1, 2, 3, 4, 5]`
	`print(List1)`
	`List1.insert(1,1.5)`
	`print(List1)`
	✧ **Output :**
	[1, 2, 3, 4, 5]
	[1, 1.5, 2, 3, 4, 5]

extend()	✧ Add elements of specified List at the end of existing List.
	✧ **Example :**
	`List1 = [1, 2, 3, 4, 5]`
	`List2 = [6, 7, 8, 9, 10]`
	`print(List1)`
	`List1.extend(List2)`
	`print(List1)`
	✧ **Output :**
	[1, 2, 3, 4, 5]
	[1, 1.5, 2, 3, 4, 5, 6, 7, 8, 9, 10]

clear()	✧ Remove all elements from List.
	✧ **Example :**
	`List1 = [1, 2, 3, 4, 5]`
	`print(List1)`
	`List1.clear()`
	`print(List1)`
	✧ **Output :**
	[1, 2, 3, 4, 5][]

remove()	✧ Remove element from List whose value is specified as an argument.
	✧ **Example :**
	`List1 = [1, 2, 3, 4, 5]`
	`print(List1)`
	`List1.remove(3)`
	`print(List1)`
	✧ **Output :**
	[1, 2, 3, 4, 5]
	[1, 2, 4, 5]

pop()	✧ Remove element from List whose index is specified as an argument.If you do not specify index than it will remove element from List which is added last.
	✧ **Example 1 :**
	`List1 = [1, 2, 3, 4, 5]`
	`print(List1)`
	`List1.pop()`
	`print(List1)`
	✧ **Output :**
	[1, 2, 3, 4, 5]
	[1, 2, 3, 4]
	✧ **Example 2 :**
	`List1 = [1, 2, 3, 4, 5]`
	`print(List1)`
	`List1.pop(2)`
	`print(List1)`
	✧ **Output :**
	[1, 2, 3, 4, 5]
	[1, 2, 4, 5]

count()	✧ Count occurrence of particular element in the List.
	✧ **Example 1 :**
	`List1 = [1, 2, 3, 4, 5, 3]`
	`print(List1.count(3))`
	✧ **Output :**
	2
	✧ **Example 2 :**
	`List1 = [1, 2, 3, 4, 5]`
	`print(List1.count(6))`
	✧ **Output :**
	0

sort()	✧ Sort elements of the List in ascending order.
	✧ **Example :**
	`List1 = [11, 23, 17, 5, 45]`
	`print(List1)`
	`List1.sort()`
	`print(List1)`
	✧ **Output :**
	[11, 23, 17, 5, 45]
	[5, 11, 17, 23, 45]

reverse()	✧ Reverse elements of the List.
	✧ Example :
	`List1 = [11, 23, 17, 5, 45]`
	`print(List1)`
	`List1.reverse()`
	`print(List1)`
	✧ Output :
	[11, 23, 17, 5, 45]
	[45, 5, 17, 23, 11]

index()	✧ Return index of the first occurrence of specified element in the List.
	✧ Example :
	`List1 = [11, 23, 11, 17, 5, 45]`
	`print(List1.index(11))`
	✧ Output :
	0

min()	✧ Returns minimum(smallest) element of the list.
	✧ Example :
	`List1 = [11, 23, 11, 17, 5, 45]`
	`print(min(List1))`
	✧ Output :
	5

max()	✧ Returns maximum(largest) element of the list.
	✧ Example :
	`List1 = [11, 23, 11, 17, 5, 45]`
	`print(max(List1))`
	✧ Output :
	45

sum()	✧ Returns sum of all elements of the list.
	✧ Example :
	`List1 = [1, 2, 3, 4, 5]`
	`print(sum(List1))`
	✧ Output :
	15

✧ **Nested and Copying Lists :**

❖ **Nested Lists**

❖ When a List is contained inside another List, it is known as Nested List. Thus a list become element of another list.

❖ **Example :**

```
Two_Wheeler = ["Activa", "Cycle"]
Three_Wheeler = ["Rickshaw","Tricycle"]
Four_Wheeler = ["Car","Bus"]
Vehicle = [Two_Wheeler, Three_Wheeler, Four_Wheeler]
print(Vehicle)
```

❖ In above example first we create three lists named Two_Wheeler, Three_Wheeler and Four_Wheeler. Next we create a List named Vehicle which contains Two_Wheeler, Three_Wheeler and Four_Wheeler list as an element.

❖ Individual element in the nested list can be accessed by specifying two indexes. First index specify the position of the element in main list and second index specify the position of element in inner list.

❖ **Example :**

```
Two_Wheeler = ["Activa", "Cycle"]
Three_Wheeler = ["Rickshaw","Tricycle"]
Four_Wheeler = ["Car","Bus"]
Vehicle = [Two_Wheeler, Three_Wheeler, Four_Wheeler]
print(Vehicle[0])          # print ['Activa', 'Cycle']
print(Vehicle[0][0])  #print  Activa
print(Vehicle[0][1])  #print  Cycle
print(Vehicle[1])          #print ['Rickshaw', 'Tricycle']
print(Vehicle[1][0])  #print  Rickshaw
print(Vehicle[1][1])  #print  Tricycle
print(Vehicle[2])          #print ['Car', 'Bus']
print(Vehicle[2][0])  #print  Car
print(Vehicle[2][1])  #print  Bus
```

✧ **Copying Lists**

❖ It is possible to create a copy of existing list into new list. In python we can copy an existing list into new list using following methods :

(1) Slicing Method :

- An existing list can be copied into new list using slicing method.
- **Example:**

```
List1 = ["A","B","C"]
List2 = List1[:]
print (List2)    # print ['A' , 'B' , 'C']
```

(2) Using list() constructor

- An existing list can be copied into new list using list() constructor.
- **Example :**

```
List1 = ["A","B","C"]
List2 = list(List1)
print (List2)    # print ['A' , 'B' , 'C']
```

(3) Using copy () method

- An existing list can be copied into new list using copy() method available in standard library named copy.
- **Example :**

```
import copy
List1 = ["A","B","C"]
List2 = copy.copy(List1)
print (List2)    # print ['A' , 'B' , 'C']
```

- We can use assignment operator (=) in order to create an alias of existing list. Alias refers to the alternate name of list.
- **Example :**

```
List1 = ["A","B","C"]
List2 = List1
print (List2)    # print ['A' , 'B' , 'C']
```

- In above example we assign List1 to List2 using assignment operator. It will not create a copy of List1 but it creates an alias of List1. It means both List1 and List2 refers to the same element in memory. If you change value of any element in List it will affect to both List.
- **Example :**

```
List1 = ["A","B","C"]
List2 = List1
List1[0]="E"
print (List1)    # print ['E' , 'B' , 'C']
print (List2)    # print ['E' , 'B' , 'C']
```

List as Arguments to Function :

- In Python, it is possible to pass list as an argument to the function. As you pass list as an argument to the function, an alias of original list is created. The function uses this alias to refer list. Any changes that are made in the list inside function will also affect original list.
- **Example :**

```
def MyListMul(MyList2):
    for i in range(0,len(MyList2)):
        MyList2[i] = MyList2[i] * 2
MyList1 = [1,2,3,4,5]
print(MyList1)          #print [1,2,3,4,5]
MyListMul(MyList1)
print(MyList1)          #print [2,4,6,8,10]
```

- In above example we pass **MyList1** as an argument to **MyListMul** function. Upon calling the function it will create an alias of MyList1 as MyList2. The function works with **MyList2** but it will also affect **MyList1**.
- In order to allow the function to works with separate list instead of the original list, you have to create a copy of original list into new list and then allow the function to work with that newly created list.

❖ Example :

```
def MyListMul(MyList2):
    MyList3 = list(MyList2)
    for i in range(0,len(MyList3)):
        MyList3[i] = MyList3[i] * 2
    print(MyList3)
MyList1 = [1,2,3,4,5]
print(MyList1)          #print [1,2,3,4,5]
MyListMul(MyList1)
print(MyList1)          #print [1,2,3,4,5]
```

1.4.2 Tuple and operations on Tuple :

- ❖ In Python Tuple is used to represent multiple values under single variable.
- ❖ Elements of the Tuple need not be of same type. It means Tuple can represents homogeneous(same) as well as non homogeneous(different) elements.
- ❖ It is a built in data type of python.
- ❖ Tuple is ordered means all the elements in the Tuple are assigned index starting from 0. Index of first element in the Tuple is 0, second element is 1 and so on.
- ❖ Tuple is un editable means you cannot add new elements into Tuple and you cannot remove elements from Tuple. You cannot update value of any element in Tuple.
- ❖ Elements of the Tuple are enclosed between round parenthesis ( ) and separated by comma.
- ❖ Tuple allows duplicate elements in it.

✧ **Creating Tuple :**

- ❖ General Syntax for creating Tuple in Python is :

```
Tuple_Name = (Element1,Element2,...,Element-N)
```

- ❖ Consider Following example which creates a Tuple named fruit and display its value:

```
fruit = ("Apple","Banana","WaterMelom","Grapes")
print(fruit)
```

- ❖ **Output :**

```
('Apple', 'Banana', 'WaterMelom', 'Grapes')
```

- ❖ Tuple can also be created using tuple() constructor. General syntax for creating tuple using tuple() constructor is given below:

```
Tuple_Name = tuple((Element1,Element2,...,Element-N))
```

- ❖ Consider Following example which creates a Tuple named fruit and display its value:

```
fruit = tuple(("Apple","Banana","WaterMelom","Grapes"))
print(fruit)
```

- ❖ **Output :**

```
('Apple', 'Banana', 'WaterMelom', 'Grapes')
```

✧ **Accessing Tuple :**

- ❖ Individual element of the Tuple can be accessed by using its index position along with name of Tuple.

- Consider following example :

```
Color = ("Red","Green","Blue","Yellow")
print(Color[1])
```

- **Output :**

```
Green
```

Note : If specified index is not present in the Tuple than it will give index out of range error.

- We can also specify negative index to access elements of tuple from last. It means -1 indicates last element of tuple and -2 indicates second last element of tuple.
- Consider following example:

```
Color = ("Red","Green","Blue","Yellow")
print(Color[-2])
```

- **Output :**

```
Blue
```

- **Check existence of element in tuple.**
- You can check if particular element exist in the Tuple or not using in operator.
- Consider following example :

```
Color = ("Red","Green","Blue","Yellow")
if "Green" in Color:
    print("Yes")
else:
    print("No")
```

- **Output :**

Yes

Iterate over tuple :

- Iterate over tuple means access each elements of tuple.
- We can access each elements of tuple using for or while loop.
- **Iterate over tuple using for loop**
- Consider following example:

```
Color = ("Red","Green","Blue","Yellow")
for x in Color:
print(x)
```

- **Output :**

```
Red
Green
Blue
Yellow
```

- We can also iterate over tuple by referencing its index positions using len() and range() method.
- Consider following example :

```
Color = ("Red","Green","Blue","Yellow")
for x in range(len(Color)):
print(Color[i])
```

- Output :

```
Red
Green
Blue
Yellow
```

- In above example len() method returns number of elements in the tuple.
- Iterate over tuple using while loop
- Consider following example:

```
Color = ("Red","Green","Blue","Yellow")
x = 0
while (x < len(Color)):
    print(Color[x])
    x = x +
```

- Output :

```
Red
Green
Blue
Yellow
```

Slicing tuple :

- Slicing tuple means access particular range of elements from tuple.
- You can access range of elements from Tuple by specifying start index and stop index separated by colon(:).
- Consider following example :

```
Color = ("Red","Green","Blue","Yellow")
print(Color[1:3])
```

- Output :

('Green', 'Blue')

- Element at Start Index is included but Element at stop index is excluded.

  You can specify negative index also to access range of elements from Tuple. Where -1 indicates index of last element,-2 indicates index of second last element and so on.
- Consider following example :

```
Color = ("Red","Green","Blue","Yellow")
print(Color[-4:-2])
```

- Output :

```
('Red', 'Green')
```

- If you don't specify start index than by default start index is considered as 0.

```
Color = ("Red","Green","Blue","Yellow")
print(Color[ :2])
```

- Output :

```
('Red', 'Green')
```

- If you don't specify stop index than by default stop index is considered as index of last element in the Tuple.

```
Color = ("Red","Green","Blue","Yellow")
print(Color[1:])
```

- Output :

```
('Green', 'Blue', 'Yellow')
```

- **Tuple are Immutable :**
  - Tuple are Immutable means it is un editable.
  - Once tuple is created you cannot add new elements into Tuple , you cannot remove elements from Tuple and you cannot update value of any element in Tuple.
- **Python Tuple Operationse :**
  - Tuple are Immutable so we cannot perform any add, delete or update operations on it.
  - However if we wish to perform any operations on tuple than we can do it by:

    Step 1 : Convert tuple to list.

    Step 2 : Perform desired operations on list.

    Step 3 : Convert list back to tuple.
  - Consider following example :

```
Color = ("Red","Green","Blue","Yellow")
ColorList = list(Color)
ColorList.append('Brown')
Color = tuple(ColorList)
print(Color)
```

  - Output :

```
('Red','Green', 'Blue', 'Yellow','Brown')
```

  - Thus, you can perform all operations of list on tuple by converting it to list.
- **Concatenation Operation :**
  - Two tuple can be concatenated using + operator in python.
  - General Syntax for concatenation operation is given below:

    Sum_Tuple = Tuple1 + Tuple2
  - It will add two tuple and store it into new tuple.
  - Consider following example :

```
Color1 = ("Red","Green","Blue","Yellow")
Color2 = ("Black","Orange","Purple")
All_Color = Color1 + Color2
print(All_Color)
```

  - Output :

```
('Red','Green', 'Blue', 'Yellow','Black','Orange','Purple')
```

  - Single element can be added to the tuple using following syntax:

    New_Tuple = Tuple1 + (Element,)

- Consider following example :

```
Color = ("Red","Green","Blue")
New_Color = Color1 + ("Brown",)
print(All_Color)
```

- Output :

```
('Red','Green', 'Blue', 'Brown')
```

Repetition Operation :

- Tuple elements can be repeated using * operator in python.
- General Syntax for repetition operation is given below:

  New_Tuple = Tuple1 * Number
- It will create new tuple by repeating elements of Tuple1 specified number of times.
- Consider following example :

```
Numbers = (1,2,3,4)
Repeated_Numbers = Numbers * 2
print(Repeated_Numbers)
```

- Output :

```
(1,2,3,4,1,2,3,4)
```

Built-In Tuple Functions and methods :

- Python supports following built in methods for tuple.

(1) count

- This method returns number of time specified element exist in the tuple.
- It counts occurrence of specified element in the tuple.
- Syntax :

```
Tuple_Name.count(Element)
```

- Example :

```
Color = ("Red","Green","Blue","Yellow","Red")
print(Color.count("Red"))
```

- Output :

  2

(2) index

- This method searches for the first occurrence of specified element in the tuple and returns its index position.
- If specified element is not found in the tuple than it generates an error.
- Syntax :

```
Tuple_Name.index(Element)
```

- Example :

```
Color = ("Red","Green","Blue","Yellow","Red")
print(Color.index("Green"))
```

- Output :

  1

- **Example :**

```
Numbers = (5,4,3,2,1)
print(sorted(Numbers))
```

- **Output :**

(1,2,3,4,5)

1.4.3 Sets and operations on Sets :

- In Python Set is used to represent multiple values under single variable.
- It is a built in data type of python.
- Set is un ordered means elements of the set are not indexed.
- Set is un editable means once set is created we cannot edit value of any element. We can add new element or delete existing element but we cannot edit existing element.
- Elements of the Set need not be of same type. It means Set can represents homogeneous(same) as well as non homogeneous(different) elements.
- Elements of the Set are enclosed between curly parenthesis { } and separated by comma.
- Set does not allow duplicate elements in it.

✧ **Create a Set, Accessing Python Sets, Delete from sets, Update sets :**

(1) Creating Set :

- General Syntax for creating Set in Python is:

```
Set_Name = {Element1,Element2,...,Element-N}
```

- Consider Following example which creates a Set named fruit and display its value:

```
fruit = {"Apple","Banana","WaterMelom","Grapes"}
print(fruit)
```

- **Output :**

```
{'Apple', 'Banana', 'WaterMelom', 'Grapes'}
```

- Set can also be created using set() constructor. General syntax for creating set using set() constructor is given below :

Set_Name = set((Element1,Element2,...,Element-N))

- Consider Following example which creates a Set named fruit and display its value:

```
fruit = set(("Apple","Banana","WaterMelom","Grapes"))
print(fruit)
```

- **Output :**

```
{'Apple', 'Banana', 'WaterMelom', 'Grapes'}
```

- As set is unordered each time you display elements of list it appears in different order.

(2) Accessing Python Sets :

- Hence, Set is un ordered you cannot access individual element of the Set using index.
- Elements of the Set can be accessed using for loop.
- Consider Following Example:

```
fruit = {"Apple","Banana","WaterMelom","Grapes"}
for x in fruit:
print(x)
```

- Output :

```
Grapes
Apple
WaterMelon
Banana
```

- We can check existence of element in the set using in operator.
- Consider following example :

```
fruit = {"Apple","Banana","WaterMelom","Grapes"}
if "Apple" in fruit:
    print("YES")
else:
    print("NO")
```

- Output :

```
YES
```

(3) Add elements in Set :

- Once set is created we can add new elements in to existing set using add method.
- General Syntax of add() method is :

```
Set_Name.add(Element)
```

- Consider Following Example :

```
fruit = {"Apple","Banana","WaterMelom"}
fruit.add("Grapes")
print(fruit)
```

- Output :

```
{'Apple', 'Banana', 'WaterMelom', 'Grapes'}
```

(4) Delete element from Set :

- Once set is created we can delete existing elements from Set.
- We can use remove() method to remove particular element from set.
- General Syntax of remove() method is:

```
Set_Name.remove(Element)
```

- Consider following example:

```
fruit = {"Apple","Banana","WaterMelom","Grapes"}
fruit.remove("Apple")
print(fruit)
```

- Output :

```
{'Banana', 'WaterMelom', 'Grapes'}
```

- If we specify an element which does not exist in the set than remove() method will generate error.
- We can also use discard() method to remove particular element from set.
- General Syntax of discard() method is:

```
Set_Name.discard(Element)
```

- Consider following example :

```
fruit = {"Apple","Banana","WaterMelom","Grapes"}
fruit.discard("Apple")
print(fruit)
```

- Output :

```
{'Banana', 'WaterMelom', 'Grapes'}
```

- If we specify an element which does not exist in the set than discard() method will not generate an error.
- Using clear() method we can delete all elements from set.
- General Syntax of clear() method is:

```
Set_Name.clear()
```

- Consider following example :

```
fruit = {"Apple","Banana","WaterMelom","Grapes"}
fruit.clear()
print(fruit)
```

- Output :

```
set()
```

- We can delete entire set using del keyword.

(5) Update Set :

- Set is un editable means once set is created we cannot edit value of any element but, we can add new elements into it.
- It is possible to add elements of other set or iterable (tuple, list or dictionary) in existing set using update() method.
- General Syntax of update() method is:

```
Set_Name.update(Iterable_Name)
```

- Consider following example :

```
fruit = {"Apple","Banana","WaterMelom","Grapes"}
vegetable = {"Potato","Tomato"}
fruit.update(vegetable)
print(fruit)
```

- Output :

```
{'WaterMelon', 'Apple', 'Grapes', 'Potato', 'Banana', 'Tomato'}
```

✧ Python Set Operations and Methods :

- Python has number of built in methods to perform various operations on set. Following are some of the useful methods of set :

Method	Purpose
add()	✧ Add new element in to set ✧ **Syntax :** Set_Name.add(Element)

	✧ **Example :** Fruit = {'Mango', 'Apple'} Fruit.add('Orange')print(Fruit) ✧ **Output :** {'Mango', 'Apple', 'Orange'}
update()	✧ Add all elements of another set or iterable in to current set. If any element common in both set or iterable than only one occurrence of the element is adde ✧ **Syntax :** Set_Name.update(Set_Name or Iterable_Name) ✧ **Example :** Fruit1 = {'Mango', 'Apple'} Fruit2 = {'Pineapple', 'Water Melon', 'Mango' } Fruit1.update(Fruit2) print(Fruit1) ✧ **Output :** {'Mango', 'Apple', 'Orange' , 'Water Melon', 'Pineapple'}
remove()	✧ Remove specified element from set. If specified element is not present in th set than it generates an error. ✧ **Syntax :** Set_Name.remove(Element) ✧ **Example :** Fruit = {'Mango', 'Apple', 'Water Melon'} Fruit.remove('Water Melon') print(Fruit) ✧ **Output :** {'Mango', 'Apple'}
discard()	✧ Remove specified element from set. If specified element is not present in th set than it will not generates an error. ✧ **Syntax :** Set_Name.remove(Element) ✧ **Example :** Fruit = {'Mango', 'Apple', 'Water Melon'} Fruit.remove('Water Melon') print(Fruit) ✧ **Output :** {'Mango', 'Apple'}
clear()	✧ Remove all elements from set. ✧ **Syntax :** Set_Name.clear()

	✧ **Example :** Fruit = {'Mango', 'Apple', 'Water Melon'} Fruit.clear()print(Fruit) ✧ **Output :** {}
copy()	✧ Returns a new set which contains all elements of current set. ✧ **Syntax :** New_Set_Name = Set_Name.copy() ✧ **Example :** Fruit = {'Mango', 'Apple', 'Water Melon'} NewFruit = Fruit.copy()print(NewFruit) ✧ **Output :** {'Mango', 'Apple', 'Water Melon'}
union()	✧ Returns union of two or more sets as a new Set. union means new Set contains elements of all Sets without duplicate. ✧ **Syntax :** Result_Set = Set_Name1.union(Set_Name2[, Set_Name3, Set_Name_N]) ✧ **Example :** Fruit1 = {'Apple', 'Banana'} Fruit2 = {'Mango', 'Pineapple', 'Apple'} Result_Fruit = Fruit1.union(Fruit2) print(Result_Fruit) ✧ **Output :** {'Apple', 'Banana', 'Mango', 'Pineapple'}
intersection()	✧ Returns intersection of two or more sets as a new set. intersection means elements common in all Sets. ✧ **Syntax :** Result_Set = Set_Name1.intersection(Set_Name2[, Set_Name3, Set_Name_N]) ✧ **Example :** Fruit1 = {'Apple', 'Banana'} Fruit2 = {'Mango', 'Pineapple', 'Apple'} Result_Fruit = Fruit1.intersection(Fruit2) print(Result_Fruit) ✧ **Output :** {'Apple'}
difference()	✧ Returns difference of two sets as a new set. Difference means elements exist in current Set but not in other Set specified as an argument.

	✧ **Syntax :** Result_Set = Set_Name1.diifference(Set_Name2) ✧ **Example :** Fruit1 = {'Apple', 'Banana', 'Cherry'} Fruit2 = {'Mango', 'Pineapple', 'Apple'} Result_Fruit = Fruit1.difference(Fruit2) print(Result_Fruit) ✧ **Output :** {'Banana', 'Cherry'}
isdisjoint()	✧ Returns true if Current Set and Specified Set has null intersection. means no common element in both Sets. ✧ **Syntax :** Set_Name1.isdisjoint(Set_Name2) ✧ **Example :** Fruit1 = {'Apple', 'Banana', 'Cherry'} Fruit2 = {'Mango', 'Pineapple', 'Apple'} print(Fruit1.isdisjoint(Fruit2)) ✧ **Output :** False
issubset()	✧ Returns true if all the elements of Current Set exist in Specified Set. ✧ **Syntax :** Set_Name1.isdisjoint(Set_Name2) ✧ **Example :** Fruit1 = {'Apple', 'Banana',} Fruit2 = {'Banana', 'Pineapple', 'Apple'} print(Fruit1.issubset(Fruit2)) ✧ **Output :** True
issuperset()	✧ Returns true if Current Set contains all the elements of Specified Set. ✧ **Syntax :** Set_Name1.isdisjoint(Set_Name2) ✧ **Example :** Fruit1 = {'Banana', 'Pineapple', 'Apple'} Fruit2 = {'Apple', 'Banana',} print(Fruit1.issubset(Fruit2)) ✧ **Output :** True

1.4.4 Dictionaries and operations on Dictionaries :

- In Python Dictionary is used to represent multiple values under single variable.
- Elements of the Dictionary need not be of same type. It means Dictionary can represents homogeneous(same) as well as non homogeneous(different) elements.
- It is a built in data type of python.
- Elements of the Dictionary are enclosed between curly parenthesis { } and separated by comma. Where Each element of the Dictionary is in the form of Key-Value pair separated by colon (:).
- In Dictionary Keys are unique but values may not be.
- Dictionary is considered to be ordered from python version 3.7.
- Once Dictionary is created you can add new elements in it as well as update value of particular element in the Dictionary. You can also delete specific or all elements of Dictionary.

✧ Creating Dictionaries :

- General Syntax for creating Dictionary in Python is:

```
Dictionary_Name =
        {Key_1:Element1,Key2:Element2,...,Key_N:Element_N}
```

- Consider Following example which creates a Dictionary named student and display its value :

```
Student = {"Name":"Yesha","RollNumber":13,"Standard":9}
print(Student)
```

- **Output :**

```
{'Name': 'Yesha', 'RollNumber': 13, 'Standard': 9}
```

✧ Accessing Items in Python Dictionaries :

- Individual element of the Dictionary can be accessed by using its key along with name of Dictionary.
- Consider Following example :

```
Student = {"Name":"Yesha","RollNumber":13,"Standard":9}
print("Name of Student Is:"+Student["Name"])
```

- **Output :**

```
Name of Student Is :Yesha
```

- If specified Key is not present in the Dictionary than it will give KeyError.
- You can also use get() method to access individual element in dictionary. Consider following example:

```
Student = {"Name":"Yesha","RollNumber":13,"Standard":9}
print("Name of Student Is: "+Student.get("Name"))
```

- **Output :**

```
Name of Student Is : Yesha
```

- In Python you can loop through each element of Dictionary using for loop. By default it will loop through each key of Dictionary and return Key. Consider following example:

```
Student = {"Name":"Yesha","RollNumber":13,"Standard":9}
for x in Student:
    print(x)
```

❖ **Output :**

```
Name
RollNumber
Standard
```

❖ If you wish to loop through each value of Dictionary than you can use **values()** method. Consider following example :

```
Student = {"Name":"Yesha","RollNumber":13,"Standard":9}
for x in Student.values():
    print(x)
```

❖ **Output :**

Yesha
13
9

❖ If you wish to loop through each value as well as key of Dictionary than you can use **items()** method. Consider following example:

```
Student = {"Name":"Yesha","RollNumber":13,"Standard":9}
for x in Student.items():
    print(x)
```

❖ **Output :**

('Name', 'Yesha')
('RollNumber', 13)
('Standard', 9)

❖ **Existence of Key or Value in the Dictionary**

❖ You can check if particular key exist in the Dictionary or not using in operator.

```
Student = {"Name":"Yesha","RollNumber":13,"Standard":9}
if "Name" in Student:
    print("Yes")
```

❖ **Output :**

```
Yes
```

❖ You can also check if particular value exist in the Dictionary or not using in operator along with values() method.

```
Student = {"Name":"Yesha","RollNumber":13,"Standard":9}
if "Yesha" in Student.values():
    print("Yes")
```

❖ **Output :**

Yes

✧ **Add, Update, Remove in Dictionaries :**

(1) Add New Item in Dictionary :

❖ New Item can be added into dictionary using following syntax :

Dictionary_Name["Key_Name"]="Element_Value"

- Consider Following Example :

```
Student = {"Name":"Yesha","RollNumber":13,"Standard":9}
Student["Division"]="C"
print(Student)
```

- **Output :**

```
{'Name': 'Yesha', 'RollNumber': 13, 'Standard': 9,'Division':'C'}
```

- We can also use update() method of Dictionary to add new Item into dictionary.
- Following is the syntax of update() method :

```
Dictionary_Name.update({"Key_Name":"Element_Value"})
```

- Consider Following Example :

```
Student = {"Name":"Yesha","RollNumber":13,"Standard":9}
Student.update({"Division":"C"})
print(Student)
```

- **Output :**

```
{'Name': 'Yesha', 'RollNumber': 13, 'Standard': 9,'Division':'C'}
```

(2) Update Existing Item in Dictionary :

- Existing Item can be updated into dictionary using following syntax:

Dictionary_Name["Key_Name"]="Element_Value"

- Consider Following Example :

```
Student = {"Name":"Yesha","RollNumber":13,"Standard":9}
Student["RollNumber"]=15
print(Student)
```

- **Output :**

```
{'Name': 'Yesha', 'RollNumber': 15, 'Standard': 9}
```

- We can also use update() method of Dictionary to update existing Item of dictionary.
- Following is the syntax of update() method :

```
Dictionary_Name.update({"Key_Name":"Element_Value"})
```

- Consider Following Example :

```
Student = {"Name":"Yesha","RollNumber":13,"Standard":9}
Student.update({"RollNumber":15})
print(Student)
```

- **Output :**

```
{'Name': 'Yesha', 'RollNumber': 15, 'Standard': 9}
```

(3) Remove Item from Dictionary :

- An item can be deleted from Dictionary using **pop()** method. It accepts Key_Name as an argument.
- Following is the syntax of pop() method :

```
Dictionary_Name.pop("Key_Name")
```

- Consider Following Example :

```
Student = {"Name":"Yesha","RollNumber":13,"Standard":9}
```

```
Student.pop("Standard")
print(Student)
```

- **Output :**

```
{'Name': 'Yesha', 'RollNumber': 15}
```

- Using **popitem()** method we can delete last inserted item from Dictionary.
- We can also use **del** keyword to delete Item with specific Key_ Name from dictionary.
- Consider following Example:

```
Student = {"Name":"Yesha","RollNumber":13,"Standard":9}
del Student["Standard"]
print(Student)
```

- **Output :**

```
{'Name': 'Yesha', 'RollNumber': 15}
```

- Entire Dictionary can also be deleted using **del** keyword.
- Consider following Example :

```
Student = {"Name":"Yesha","RollNumber":13,"Standard":9}
del Student
print(Student)
```

- **Output :**

NameError: Name Student is not defined

- We can also use **clear()** method to remove all Items from dictionary.
- Consider following Example :

```
Student = {"Name":"Yesha","RollNumber":13,"Standard":9}
Student.clear()
print(Student)
```

- **Output :**

{}

Built-In Dictionary Methods and functions :

- Python has number of built in methods to perform various operations on Dictionary. Followin are some of the useful methods of Dictionary :

Method	Purpose
get()	✧ Returns value of key specified as an argument. ✧ You can also specify default value for the key that does not exist in the dictionar ✧ **Syntax :** Value = Dictionary_Name.get(Key_Name[,DefaultValue]) ✧ **Example1 :** Student = {"Name":"Yesha", "RollNumber":13} Value = Student.get("Name") print(Value)

	✧ **Output :** Yesha ✧ **Example2 :** Student = {"Name":"Yesha", "RollNumber":13} Value = Student.get("Standard",9) print(Value) ✧ **Output :** 9
keys()	✧ Returns a list which contains keys of Dictionary. ✧ **Syntax :** List_Name = Dictionary_Name.keys() ✧ **Example :** Student = {"Name":"Yesha", "RollNumber":13} Key_List = Student.keys() print(Key_List) ✧ **Output :** dict_keys(['Name', 'RollNumber'])
values()	✧ Returns a list which contains values of Dictionary. ✧ **Syntax :** List_Name = Dictionary_Name.values() ✧ **Example :** Student = {"Name":"Yesha", "RollNumber":13} Value_List = Student.values() print(Value_List) ✧ **Output :** dict_values(['Yesha', 13])
items()	✧ Returns a list of tuple in which each tuple contains name value pair of dictionary. ✧ **Syntax :** List_Name = Dictionary_Name.items() ✧ **Example :** Student = {"Name":"Yesha", "RollNumber":13} Item_List = Student.items() print(Item_List) ✧ **Output :** dict_items([('Name', 'Yesha'), ('RollNumber', 13)])
clear()	✧ Removes all elements from Dictionary. ✧ **Syntax :** Dictionary_Name.clear()

	✧ **Example :** Student = {"Name":"Yesha", "RollNumber":13} Student.clear() print(Student) ✧ **Output :** {}

pop()	✧ Removes element with specified key from Dictionary.
	✧ **Syntax :**
	Dictionary_Name.pop("Key_Name")
	✧ **Example :**
	`Student = {"Name":"Yesha","RollNumber":13,"Standard":9}`
	`Student.pop("Standard")`
	`print(Student)`
	✧ **Output :**
	`{'Name': 'Yesha', 'RollNumber': 15}`

popitem()	✧ Removes key : value pair from Dictionary which is inserted last.
	✧ **Syntax :**
	Dictionary_Name.popitem ()
	✧ **Example :**
	`Student = {"Name":"Yesha","RollNumber":13,"Standard":9}`
	`Student.popitem()`
	`print(Student)`
	✧ **Output :**
	`{'Name': 'Yesha', 'RollNumber': 15}`

update()	✧ Updates Dictionary with new {Key : Value} pair. Using this method we can ad new element as well as update existing element.
	✧ **Syntax :**
	Dictionary.update('Key_Name','Value')
	✧ **Example :**
	`Student = {"Name":"Yesha","RollNumber":13}`
	`Student.update('Standard':9)`
	`print(Student)`
	✧ **Output :**
	`{'Name': 'Yesha', 'RollNumber': 15, 'Standard':9}`

copy()	✧ Returns new dictionary which contains copy of current Dictionary. ✧ **Syntax :** Dictionary_Copy = Dictionary_Name.copy()

✧ **Example :**

```
Student = {"Name":"Yesha","RollNumber":13,"Standard":9}
Student_Copy = Student.copy()
print(Student_Copy)
```

✧ **Output :**

```
{'Name': 'Yesha', 'RollNumber': 15, 'Standard':9}
```

1.5 Array in Python :

- Array is a collection of elements of same data type. Array is used to store multiple value using single variable.
- Array is a sequential. It means memory is allocated sequentially for all the elements of array.
- In python we have to import module named **array** in order to use array.

  import array
- General Syntax for creating array in python is :

  `Array_Name = array.array('Type_Code'[,[Element_List]])`

  Here,

  Type_Code represents type of the value contained by array.

  Element_List is optional. If you want to initialize array at the time of creation than you can specify Element_List.
- You can specify following Type_Code :

Type_Code	Meaning
b	Signed Char
B	Unsigned Char
u	Unicode Char
h	Signed Short
H	Unsigned Short
i	Signed Int
I	Unsigned Int
l	Signed Long
L	Unsigned Long
f	float
d	double

- **Consider Following Example :**

```
import array
myArray = array.array('i',[1,2,3,4,5])
print(myArray[1]) # It will display 2
```

❖ We can display Type Code of array as well as Size of array using following code :

```
import array
a = array.array('i',[1,2,3,4])
print("Type Code Of Array Is:"+a.typecode)
print("Number Of Elements in Array is:"+str(a.itemsize))
```

1.5.1 Operations on Arrays :

❖ Various operations that can be performed on array are given below :

(1) append

- It will insert new element at the end of array.
- **Syntax :**

  Array_Name.append(Value)
- **Example :**

```
import array
a = array.array('i',[1,2,3,4])
print(a)
a.append(5)
print(a)
```

- **Output :**

  array('i', [1, 2, 3, 4])

  array('i', [1, 2, 3, 4, 5])

(2) insert

- It will insert new element at specified index position.
- **Syntax :**

  Array_Name.insert(Index,Value)
- **Example :**

```
import array
a = array.array('i',[1,2,3,4])
print(a)
a.insert(1,7)
print(a)
```

- **Output :**

  array('i', [1, 2, 3, 4])

  array('i', [1, 7, 2, 3, 4])

(3) pop

- It will remove element from specified index position, If Specified index not found than it will give **pop index out of range** error.
- **Syntax:**

  Array_Name.pop(Index)

- **Example :**

```
import array
a = array.array('i',[1,2,3,4])
print(a)
a.pop(1)
print(a)
```

- **Output :**

array('i', [1, 2, 3, 4])
array('i', [1, 3, 4])

(4) remove

- It will remove first occurrence of the specified element . If Specified element not found than it will give **element not in array** error.
- **Syntax :**

Array_Name.remove(Value)

- **Example :**

```
import array
a = array.array('i',[1,2,3,4,2])
print(a)
a.remove(2)
print(a)
```

- **Output :**

array('i', [1, 2, 3, 4,2])
array('i', [1, 3, 4,2])

(5) Index

- It will return index position of the first occurrence of the specified element . If Specified element not found than it will give **element not in array** error.
- **Syntax :**

Array_Name.index(Value)

- **Example :**

```
import array
a = array.array('i',[1,2,3,4,2])
print(a)
print(a.index(2))
```

- **Output :**

array('i', [1, 2, 3, 4,2])
1

(6) reverse

- It will reverse the order of elements of array.
- **Syntax :**

Array_Name.reverse()

- **Example :**

```
import array
a = array.array('i',[1,2,3,4,5])
print(a)
a.reverse()
print(a)
```

- **Output :**

array('i', [1, 2, 3, 4,5])
array('i', [5, 4, 3, 2,1])

(7) count

- It will count occurrence of specified element in array.
- **Syntax :**

Array_Name.count(Value)

- **Example :**

```
import array
a = array.array('i',[1,2,3,4,5,2,3,2])
print(a)
print(a.count(2))
```

- **Output :**

array('i', [1, 2, 3, 4,5,2,3,2])
3

(8) extend

- It will append all the elements of specified iterable at the end of array.
- **Syntax :**

Array_Name.extend(Iterable_Name)

- **Example :**

```
import array
a = array.array('i',[1,2,3,4,5])
b = [10,11,12,13,14,15]
print(a)
a.extend(b)
print(a)
```

- **Output :**

array('i', [1, 2, 3, 4,5])
array('i', [1, 2, 3, 4,5,10,11,12,13,14,15])

1.5.2 Arrays Vs List :

Array	List
Array is a collection of elements of same data type.	List is a collection of elements of either same or different data type.
In order to use array we need to import module named array.	In order to use list there is no need to import any module.
Array is preferred when number of elements are more.	List is preferred when number of elements are few.
Processing of array element is faster.	Processing of list element is slower.