# Unit 4
# Lists, Tuples, Dictionaries

*Lists: list operations, list slices, list methods, list loop, mutability, aliasing, cloning lists, list parameters; Tuples: tuple assignment, tuple as return value; Dictionaries: operations and methods; advanced list processing - list comprehension; Illustrative programs: selection sort, insertion sort, merge sort, histogram.*

## 4.1 LISTS

One of the basic data structure in Python is sequence. There are six built-in types of sequence - strings, Unicode strings, lists, tuples, buffers, and xrange objects. The most common are List and Tuple. Several operations can be done using sequences. For example; indexing, slicing, adding, multiplying, and checking for membership. In addition, Python has built-in functions for finding the length of a sequence and for finding its largest and smallest elements. A list is an ordered set of values, where each value is identified by an index. The values in the list are called its elements. Each element of a sequence is assigned a number - its position or index. The first index is zero, the second index is one, and so forth.

### 4.1.1 Creation and Accessing List

The list can be created by putting values (items/elements) separated by commas between square brackets [ ]. List is one of the most frequently used and very versatile data type used in Python. Like most programming languages, Python's index also starts from 0. The values in the list can be of any data type. In the simplest form the syntax of list can be given as L = [expression for variable in sequence].

Syntax for List :

<List_Name> = [<Value1>, <Value2>....., <ValueN>]

Examples

```
# empty list
EMP_list = [];
# list of integers
Nos_list = [1, 2, 3];
# list with mixed datatypes
Hetro_list = [1, "Hello", 3.4];
```

Nested lists are also acceptable. The line 5 has nested list. The list can be accessed using the index value. In the program 1, lines 7 to 10 access the values in the list declared in line 3 to 5. In line 8; the range takes two arguments and returns a list that contains all the elements from the first to the second, including the first but not including the second. Any integer expression can be used as an index; line 10. There are two ways of accessing the lists; List Index and Negative Indexing. If an index has a negative value, it counts backward from the end of the list; line 10. The index -1 refers to the last item, -2 to second last item and so on.

**PROGRAM 1**

**Simple Example for Lists**

```
1. # List Example in Python
2.
3. Hetro_List = ['physics', 'chemistry', 1997, 2000]
4. Nos_List = [1, 2, 3, 4, 5 ]
5. Nested_List = ["X", "Y", [10,4,5]]
6.
7. print "Hetro_List[0]: ", Hetro_List[0]
8. print "Nos_List[1:3]: ", Nos_List[1:3]
9. print "Nested_List[2]: ", Nested_List[7-5]
10. print "Nos_List[-2]: ", Nos_List[-2]
```

**EXECUTION**

```
sh-4.3$ python program 1.py
Hetro_List[0]: physics
Nos_List[1:3]: [2, 3]
Nested_List[2]: [10,4,5]
Nos_List[-2]: 4
```

### 4.1.2 List Operations

Two operators + and * is allowed in list. + concatenates the lists and * repeats the list elements a given number of time.

**PROGRAM 2**

**Simple Example for List operations**

```
1. # List Operations Example
2.
3. Nos_List = [1, 2, 3]
4. Char_List = ["a", "b"]
5.
6. #add two lists
7. Add_List = Nos_List + Char_List
8.
9. #Append same elements n no of times
```

```
10. Repeat_List = Char_List * 3
11.
12. print "Add_List : ", Add_List
13. print "Repeat_List : ", Repeat_List
```

**EXECUTION**

```
sh-4.3$ python program 2.py
Add_List : [1, 2, 3, 'a', 'b']
Repeat_List : ['a', 'b', 'a', 'b', 'a', 'b']
```

### 4.1.3 List Slices

A subset of elements of a list is called a slice of list. Selecting a list slice is similar to selecting a character in a string.

**PROGRAM 3**

**Simple Example for List Slicing**

```
1. # List Slicing Example
2.
3. Nos_List = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
4.
5. # Different slicing of list
6. print "Elements 3rd to 5th : ", Nos_List[2:5]
7. print "Elements beginning upto -5th index : ", Nos_List[:-5]
8. print "Elements 6th to end : ", Nos_List[5:]
9. print "Elements beginning to end : ", Nos_List[:]
```

**EXECUTION**

```
sh-4.3$ python program 3.py
Elements 3rd to 5th : [3, 4, 5]
Elements beginning to -5th index : [1, 2, 3, 4, 5, 6]
Elements 6th to end : [6, 7, 8, 9, 10, 11]
Elements beginning to end : [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
```

### 4.1.4 List Methods

The list data type has some methods. Methods that are available for the list object in Python programming are tabulated below. The following table gives the result of every method for the considered list. In general the methods are accessed as ListVariable.MethodName()

| S. No. | Method | Description |
|---|---|---|
| 1. | append(E) | Add an element(E) to the end of the list |
| 2. | extend(seq) | Add all elements of a list to the another list(seq) |
| 3. | insert(ind,E) | Insert an element (E) at the defined index(ind) |
| 4. | remove(E) | Removes the first occurrence of element (E) from the list |
| 5. | pop(ind) | Removes and returns an element at the given index (ind) |
| 6. | index(E) | Returns the index of the first matched element with E |
| 7. | count(E) | Returns the count of number of elements (E) in the list |
| 8. | sort() | Sort items in a list in ascending order |
| 9. | reverse() | Reverse the order of items in the list |
| 10. | copy() | Returns a shallow copy of the list |

Table 4.1 Methods in List

Illustration of Methods in List

Considering the list; Nos_List = [1, 2, 3, 4, 5, 6, 7, 8, 9];

| S.No. | Method | Nos_List |
|---|---|---|
| 1. | Nos_List.append(5) | [1, 2, 3, 4, 5, 6, 7, 8, 9, 5] |
| 2. | Nos_List.extend([5,6]) | [1, 2, 3, 4, 5, 6, 7, 8, 9, 5, 5, 6] |
| 3. | Nos_List.insert(4,5) | [1, 2, 3, 4, 5, 5, 6, 7, 8, 9, 5, 5, 6] |
| 4. | Nos_List.remove(3) | [1, 2, 4, 5, 5, 6, 7, 8, 9, 5, 5, 6] |
| 5. | C=Nos_List.pop(6) | [1, 2, 4, 5, 5, 6, 8, 9, 5, 5, 6]; C=7 |
| 6. | Ind=Nos_List.index(5) | Ind=3 |
| 7. | Cnt=Nos_List.count(5) | Cnt=4 |
| 8. | Nos_List.reverse() | [6, 5, 5, 9, 8, 6, 5, 5, 4, 2, 1] |
| 9. | Nos_List.sort() | [1, 2, 4, 5, 5, 5, 5, 6, 6, 8, 9] |
| 10. | ListCopy=Nos_List.copy() | ListCopy=[1, 2, 4, 5, 5, 5, 5, 6, 6, 8, 9] |

Table 4.2 Illustration of Methods in List

### 4.1.5 List Functions

The functions are different from methods. The functions return a datatype and definitely take up one or more arguments. Most of the functions are common to all the compound data types. The tables 4.3 and 4.4 explain and illustrate in detail about the functions available for the list data type.

| S.No. | Function | Description |
|---|---|---|
| 1. | cmp(list1, list2) | Compares elements of both lists. |
| 2. | len(list) | Gives the total length of the list. |
| 3. | max(list) | Returns item from the list with max value. |
| 4. | min(list) | Returns item from the list with min value. |
| 5. | list(seq) | Converts a tuple into list. |
| 6. | range(N)<br>range(Start,End,StepSize) | Creates a list consecutive N nos starting from 0.<br>Creates a list from Start no to End no ; skipping in between StepSize |

Table 4.3 Functions in List

| S.No. | Input | Function | Output |
|---|---|---|---|
| 1. | list1 = [123, 'xyz']<br>list2 = [456, 'abc']<br>list3= [456, 'abc'] | print cmp(list1, list2)<br>print cmp(list2, list1)<br>print cmp(list3, list1) | – 1<br>1<br>0 |
| 2. | Nos_List = [1, 2, 3, 4, 5, 6, 7, 8, 9] | len(Nos_List) | 9 |
| 3. | Nos_List = [1, 2, 3, 4, 5, 6, 7, 18, 9] | max(Nos_List) | 18 |
| 4. | Nos_List = [1, -32, 3, 4, 5, 6, 7, 8, 9] | min(Nos_List) | – 32 |
| 5. | aTuple = (123, 'xyz', 3) | aList = list(aTuple)<br>print "List elements:", | List elements:<br>aList [123, 'xyz', 3] |
| 6. | 5<br>1, 10, 2 | L= range(5);<br>K=range(1,10,2);<br>print "L = ", L<br>print "K = ", K | L=[1, 2, 3, 4, 5]<br>K= [1, 3, 5, 7, 9] |

Table 4.4 Illustration of Functions in List

### 4.1.6 List Loop

Looping in a list is used to access every element in the list.
There are two types of looping.
Normal for-in loop as specified in line 8 & 9
Range of Length loop as specified in line 13 & 14

**PROGRAM 4**

**Simple Example for List Looping**

```
1. # List Looping Example
2.
3. colors_list= ["red", "green", "blue", "purple"]
4. months_list= ["Jan", "Feb", "Mar", "Apr","May"]
5.
6. print "Colors in the List"
7. # Normal Looping
8. for i in range(len(colors_list)):
9. print(colors_list[i])
10.
11. # Range Looping
12. print "Months in the List"
13. for month in months_list:
14. print(month)
```

**EXECUTION**

```
sh-4.3$ python program 4.py
Colors in the List
red
green
blue
purple
Months in the List
Jan
Feb
Mar
Apr
May
```

The lines 8 and 9 are like normal for loop; where each element in the list are accessed using the index value. The lines 13 and 14 are special case; where every element in the list of months_list is taken. The iteration stops when all the elements in the list are exhausted.

### 4.1.7 List Mutability

Mutability is the ability for certain types of data to be changed without entirely recreating it. List is a mutable data type; which means we can change their elements.

**PROGRAM 5**

**Simple Example for List Mutability**

```
1. # List Mutable Example
2.
3. colors_list = ["red", "green", "blue", "purple"]
4. print "Original List : ", colors_list
5. #red is changed to pink
6. colors_list[0]="pink"
7.
8. #blue is changed to orange
9. colors_list[-2]="orange"
10. print colors_list
```

**EXECUTION**

```
sh-4.3$ python program 5.py
Original List : ["red", "green", "blue", "purple"]
['pink', 'green', 'orange', 'purple']
```

### 4.1.8 Aliasing

Same object or list being referenced by two variables means aliasing. For example list Nos_List is referenced by another variable temp_list; changes made in temp_list will be reflected in Nos_List too. It is safer to avoid aliasing when you are working with mutable objects.

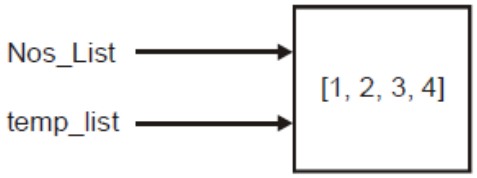


Figure 4.2 List Aliasing

**PROGRAM 6**

**Simple Example for List Aliasing**

```
1. # List Aliasing Example
2.
3. Nos_List = [1, 2, 3, 4]
4.
5. # temp_list is Alias
6. temp_list = Nos_List
7.
8. print "temp_list: " , temp_list
9. # Alias changed will change original too
10. temp_list.append(5)
11.
12. print "Nos_List after change in temp: " , Nos_List
```

**EXECUTION**

```
sh-4.3$ python program 6.py
temp_list: [1, 2, 3, 4]
Nos_ List after change in temp: [1, 2, 3, 4, 5]
```

### 4.1.9 Cloning Lists

Cloning is different from aliasing. Aliasing just creates another name for the same list. Cloning creates a new list with same values under another name. Taking any slice of a list creates a new list.

**PROGRAM 7**

**Simple Example for List Cloning**

```
1. # List Cloning Example
2.
3. fruits = ["apples","pear","grapes","orange","mango"]
4.
5. # Aliasing list gives it another name
6. Fruit_List= fruits
7.
8. # Copying a complete slice CLONES and creates a new list
9. CloneFruit= Fruit_List[:]
10. # Doesnt affect original list
11. CloneFruit.append("Banana")
```

```
12.
13. # Affects the original list
14. Fruit_List[1] = "Lemon"
15.
16. # Display the original, alias and clone
17. print "Original List : ", fruits
18. print "Alias List : ", Fruit_List
19. print "Clone List : ", CloneFruit
```

**EXECUTION**

```
sh-4.3$ python program 7.py
Original List : ['apples', 'Lemon', 'grapes', 'orange', 'mango']
Alias List : ['apples', 'Lemon', 'grapes', 'orange', 'mango']
Clone List : ['apples', 'pear', 'grapes', 'orange', 'mango', 'banana']
```

The below figure explains how alias and list refer the values in the same memory location; whereas the clone refers to same value but they are stored in different memory location. The changes made to the list via List Name or Alias Name; both will affect the same list. However the clone is a separate variable and hence the changes made to a clone will never affect the original list.

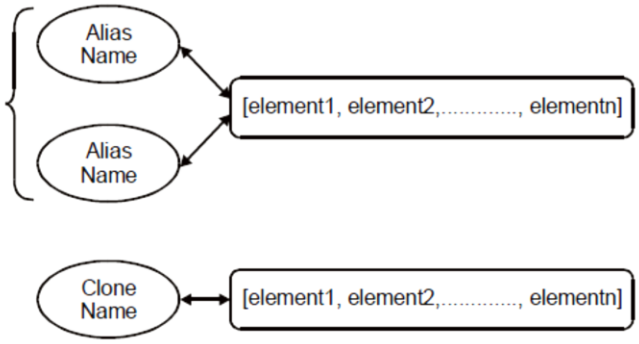


Figure 4.3 Difference between Cloning and Aliasing

4.1.10 List Deletion

The elements in the list can be deleted by using del statement. del works similar to slicing of lists. In slicing a new list is created with the elements sliced from the input list and in del the sliced elements are removed from the input list. The input list is changed after execution of del statement. The general syntax of del statement is:

Syntax:

del <list_name>[index/slice]

**PROGRAM 8**

**Simple Example for List Deletion**

```
1. # List Deletion in Python
2.
3. Nos_List = [1, 2, 3, 4, 5, 6, 7, 8, 9];
4.
```

```
5. # Deletion with index
6. del Nos_List[0]
7. print "Nos_List after deleting first element: ", Nos_List
8. del Nos_List[-1]
9. print "Nos_List after deleting last element: ", Nos_List
10.
11. # Deletion with slices
12. del Nos_List[1:5]
13. print "Nos_List after deleting first 1st to 4th elements: ", Nos_List
```

**EXECUTION**

```
sh-4.3$ python program 8.py
Nos_List after deleting first element: [2, 3, 4, 5, 6, 7, 8, 9]
Nos_List after deleting last element: [2, 3, 4, 5, 6, 7, 8]
Nos_List after deleting first 1st to 4th elements: [2, 7, 8]
```

### 4.1.11 List Parameters

Lists can be passed as arguments to functions. The list arguments are always passed by reference only. Hence changes made to list in the scope of function are permanent and visible outside the function block too. The program 9 implements stack using list data type. Stack is an abstract data type in which elements pushed down. The first element is below the second and second is below third and so on. To remove the first element; the second element has to be removed. Basically a stack follows first in first out strategy. In the below example the variable stack and Nos_Stack are aliases of the list. Hence after every push an element is added to the list.

**PROGRAM 9**

**Simple Example for List Parameters**

```
1. # List parameters in Python
2.
3. # Function to add element at the end
4. def push(stack, No, Top):
5. stack.append(No)
6. Top=Top+1
7. return
8.
9. # Function to remove and get element at the end
10. def pop():
11. return Nos_Stack.pop(Top)
12.
13. Nos_Stack = []
14. Top=0
15.
16. # Creating a list of 4 elements
```

```
17. push(Nos_Stack, 10, Top)
18. push(Nos_Stack, 13, Top)
19. push(Nos_Stack, 89, Top)
20. push(Nos_Stack, -9, Top)
21.
22. print "Nos_Stack after pushes : ", Nos_Stack
23. Top=Top-1
24.
25. # Removing recent element
26. top_item = pop()
27. print "Nos_Stack after pop : ", Nos_Stack
```

**EXECUTION**

```
sh-4.3$ python program 9.py
Nos_Stack after pushes : [10, 13, 89, -9]
Nos_Stack after pop : [10, 13, 89]
```

### 4.2 TUPLES

Tuples are similar to lists. They are immutable sequences. The elements in the tuple cannot be modified as in lists. The differences between tuples and lists are, the tuples cannot be modified like lists and tuples use parentheses, whereas lists use square brackets.

Syntax:

for Tuples

<Tuple_Name> = <Value1>, <Value2>....., <ValueN>

(or)

<Tuple_Name> = (<Value1>, <Value2>....., <ValueN>)

#### 4.2.1 Creation and Accessing Tuple

Tuples can be created by putting different values separated by comma; the parentheses are optional.

**Examples**

```
# empty Tuple
    EMP_ Tuple = ()
# Tuple of integers
    Nos_ Tuple = (1, 2, 3)
# Tuple with mixed datatypes
    Hetro_ Tuple = (1, "Hello", 3.4)
```

**PROGRAM 10**

**Simple Example for Tuples**

```
1. # Tuples Example in Python
2.
3. # Simple Tuple
```

```
4. Nos_ Tuple = (1, 2, 3)
5. print Nos_ Tuple
6.
7. # Nested Tuple
8. Nested_tuple = ("string", ['list', 'in', 'tuple'],
('tuple', 'in', 'tuple'))
9. print Nested_tuple[0]
10. print (Nested_tuple[1])
11. print Nested_tuple[2]
12.
13. # Mixed Tuple creation
14. Mixed_tuple = 3, "kg", "apples"
15. print(Mixed_tuple)
16. Nos, Meseaure, What = Mixed_tuple
17.
18. print(Nos)
19. print(Meseaure)
20. print(What)
```

**EXECUTION**

```
sh-4.3$ python program 10.py
(1, 2, 3)
string
['list', 'in', 'tuple']
('tuple', 'in', 'tuple')
(3, 'kg', 'apples')
3
kg
apples
```

### 4.2.2 Operations in Tuples

Two operators + and * is allowed in tuples. + concatenates the tuples and * repeats the tuple elements a given number of times.

**PROGRAM 11**

**Simple Example for Tuple Opeartions**

```
1. # Tuple Operations Example
2.
3. Nos_tuple = (1, 2, 3, 4,7, 8,9)
4. Rep_tuple = ('a' , ) * 5
5.
6. #add two lists
7. New_tuple = Nos_tuple + Rep_tuple
```

```
8.
9. print "Add Operation: ", New_tuple
10. print "Repeat Operation : ", Rep_tuple
```

**EXECUTION**

```
sh-4.3$ python program 11.py
Add Operation: (1, 2, 3, 4, 7, 8, 9, 'a', 'a', 'a', 'a', 'a')
Repeat Operation : ('a', 'a', 'a', 'a', 'a')
```

### 4.2.3 Slicing and Indexing of Tuples

Slicing of tuples is similar to lists. The below example is same as example 1.

**PROGRAM 12**

**Simple Example for Tuple Slicing and Indexing**

```
1. # Tuple Example in Python
2.
3. Hetro_Tuple = ('physics', 'chemistry', 1997, 2000)
4. Nos_Tuple = (1, 2, 3, 4, 5)
5. Nested_Tuple = ("X", "Y", (10, 4, 5))
6.
7. # Different types of Indexing - With index
8. print "Hetro_Tuple[0]: ", Hetro_Tuple[0]
9.
10. # Different types of Indexing - With expression
11. print "Nested_Tuple[2]: ", Nested_Tuple[7-5]
12.
13. # Different types of Indexing-With negative nos; backwards
14. print "Nos_Tuple[-2]: ", Nos_Tuple[-2]
15.
16. # Slicing
17. print "Nos_Tuple[1:3]: ", Nos_Tuple[1:3]
```

**EXECUTION**

```
sh-4.3$ python program 12.py
Hetro_Tuple[0]: physics
Nested_Tuple[2]: (10, 4, 5)
Nos_Tuple[-2]: 4
Nos_Tuple[1:3]: (2, 3)
```

### 4.2.4 Deleting and Updating Tuples

Tuples are immutable; the elements cannot be changed. In lists the elements can be deleted. In tuples elements cannot be deleted. However the entire tuple can be deleted using del. Similarly the elements cannot be modified in a tuple. However if the tuple has a mutable data element; it can be changed. Programs 13 and 14 explain in detail the deletion and updation of tuples.

**PROGRAM 13**

**Simple Example for Updating Tuples**

```
1. # Example of Updating Tuples
2.
3. #A tuple is immutable; however the list inside a tuple can be changed
4. Nested_tuple = ("string", ['list', 'in', 'tuple'], ('tuple', 'in', 'tuple'))
5. Nested_tuple[1][2] = 'changed'
6. print Nested_tuple
7.
8. #Try Changing an element
9. C_tuple = ('c','o','m','p','u','t','e')
10. C_tuple[0] = 'o'
```

**EXECUTION**

```
sh-4.3$ python program 13.py
('string', ['list', 'in', 'changed'], ('tuple', 'in', 'tuple'))
Traceback (most recent call last):
File "main.py", line 10, in <module>
C_tuple[0] = 'o'
TypeError: 'tuple' object does not support item assignment
```

**PROGRAM 14**

**Simple Example for Deleting Tuples**

```
1. # Example of Deletion of Tuples
2.
3. #A tuple is immutable; however the list inside a tuple can be changed
4. Nested_tuple = ("string", ['list', 'in', 'tuple'], ('tuple', 'in', 'tuple'))
5. del Nested_tuple[1][2]
6. print Nested_tuple
7.
8. # Deleting the tuple
9. del Nested_tuple
10.
11. # Trying to delete one element
12. C_tuple = ('c','o','m','p','u','t','e')
13. del C_tuple[0]
```

**EXECUTION**

```
sh-4.3$ python program 14.py
('string', ['list', 'in'], ('tuple', 'in', 'tuple'))
Traceback (most recent call last):
File "main.py", line 13, in <module>
del C_tuple[0]
TypeError: 'tuple' object doesn't support item deletion
```

### 4.2.5 Tuple Functions

Functions of tuple are also similar to list data type. The tuples are immutable lists and are represented by parentheses instead of square brackets. The methods of tuple are also similar to list. The tables 4.1 and 4.2 explain in detail the methods applicable for list. The same holds good for tuples too.

| S.No. | Function | Description |
|---|---|---|
| 1. | cmp(tuple1, tuple2) | Compares elements of both tuples. |
| 2. | len(tuple) | Gives the total length of the tuple. |
| 3. | max(tuple) | Returns item from the tuple with max value. |
| 4. | min(tuple) | Returns item from the tuple with min value. |
| 5. | tuple(seq) | Converts a sequence into tuple. |
| 6. | sum(tuple, S) | Return the sum of all elements and S |
| 7. | sorted(tuple) | Sorts the tuple in ascending or descending order |

Table 4.5 Functions in Tuple

| S.No. | Input | Function | Output |
|---|---|---|---|
| 1. | Tup1 = (123, 'xyz')<br>Tup2 = (456, 'abc')<br>Tup3 = (456, 'abc') | print cmp(Tup1, Tup2)<br>print cmp(list2, list1)<br>print cmp(list3, list1) | – 1<br>1<br>0 |
| 2. | Nos_Tup = (1, 2, 3, 4, 5, 6, 7, 8, 9) | len(Nos_Tup) | 9 |
| 3. | Nos_Tup = (1, 2, 3, 4, 5, 6, 7, 18, 9) | max(Nos_Tup) | 18 |
| 4. | Nos_Tup = (1, -32, 3, 4, 5, 6, 7, 8, 9) | min(Nos_Tup) | – 32 |
| 5. | aList = [123, 'xyz', 3] str = 'compute' | aTuple = tuple(aList)<br>sTuple = tuple(str) | aTuple = (123, 'xyz', 3)<br>sTuple = ('c', 'o', 'm', 'p', 'u', 't', 'e') |
| 6. | Nos_Tup = (1, -32, 3, 4, 5, 6, 7, 8, 9) | sum(Nos_Tup)<br>sum(Nos_Tup,2) | 11<br>13 |
| 7. | Nos_Tup = (1, -32, 3, 4, 5, 6, 7, 8, 9) | sorted(Nos_Tup)<br>sorted(Nos_Tup,<br>reverse=true) | [–32, 1, 3, 4, 5, 6, 7, 8, 9]<br><br>[9, 8, 7, 6, 5, 4, 3, 1, –32] |

Table 4.6 Illustration of Functions in Tuples

### 4.2.6 Tuple Assignment

Swapping of values in two variables need three statements and one temporary variable as below;
Temp = A
A = B
B = Temp
Tuples provide a very easy way to do swapping; as specified in line 5 in the below program. This form of assignment is called tuple assignment. The tuple assignment can be done for swapping any number of values as in line 10. The number of variables in left and right of assignment operator must be equal. A single assignment to paralleling assign values to all elements of a tuple is the major benefit of the tuple assignment.

**PROGRAM 15**
**Simple Example for Tuple assignment**

```
1. # Tuple Swapping in Python
2.
3. A = 100
4. B = 345
5. C = 450
6. print "A & B", A, "&", B
7.
8. # Tuple assignment
9. A, B = B, A
10. print "A & B after tuple assignment : ", A, "&", B
11.
12. # Tuple assignment can be done for any no of variables
13. A, B, C = C, A, B
14. print "Tuple assignment more values : ",A,"&",B,"&"C
```

**EXECUTION**

```
sh-4.3$ python program 15.py
A & B : 100 & 345
A & B after tuple assignment : 345 & 100
Tuple assignment more values : 450 & 345 & 100
```

### 4.2.7 Tuple as Return Value

Functions can return tuples. The swapping in Example 15 can be achieved by function in line 3. The function swap returns a tuple of values and in line 12 the swapped tuple values are assigned to the variables A, B and C.

**PROGRAM 16**
**Simple Examples for Returning values**

```
1. # Tuple as Return Value in Python
2.
3. def swap(a,b,c):
```

```
4. return c, b , a
5.
6. A = 100
7. B = 345
8. C = 765
9. print "A,B,C : ", A, " ", B, ",", C
10.
11. # Calling function swap
12. A, B, C = swap(A, B, C)
13. print "After Swap : ", A, ",", B, ",", C
```

**EXECUTION**

```
sh-4.3$ python program 16.py
A,B,C : 100 345 , 765
After Swap : 765 , 345 , 100
```

The program 17 is another example for tuple as return value. The values of quotient and remainder are returned as a tuple. The function mod_div in line 3, returns a tuple that consists of quotient and remainder. The input sec is converted to HH::MM::SS format using the mod_div function.

**PROGRAM 17**

```
1. # Another example for Tuple as return value
2.
3. def mod_div(x, y):
4. # Find modulus and quotient
5. quotient = x/y
6. remainder = x % y
7. return quotient, remainder
8.
9. # Input the seconds and get the hours minutes and seconds
10. sec = 4234
11. minutes, seconds = mod_div(sec, 60)
12. hours, minutes = mod_div(minutes, 60)
13. print ('%d seconds = %d hrs::%d min::%d sec' % (sec, hours, minutes, seconds))
```

**EXECUTION**

```
sh-4.3$ python program 17.py
4234 seconds = 1 hrs::10 min::34 sec
```

## 4.3 DICTIONARIES

Dictionary is one of the compound data type like string, list and tuple. In dictionary the index can be any immutable datatype; mostly string or integers. Every element in a dictionary is the key-value pair.

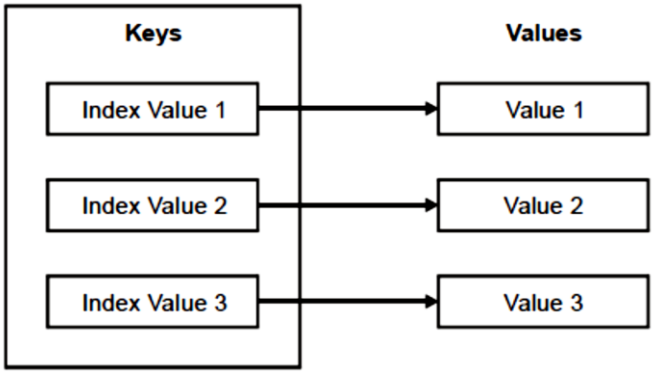


Figure 4.4 Structure of Dictionary

### 4.3.1 Creation and Accessing

There are two ways to create dictionaries. It can be created by specifying the key and value separated by a colon (:) and the elements separated by commas. The entire set of elements must be enclosed by curly braces. The benefit of dictionary is to map or associate the keys to the values. Another way is to use dict() constructor that uses a sequence to create dictionary. Note the line 5 in program 13.

Examples

```
# empty dictionary
    EMPTY_dict = {}
# Dictionary with integer keys
    Nos_dict = {1: 'fruit', 2: 'vegetable', 3: 'fish'}
# Keys as mixed datatypes
    Hetro_dict = {'name': 'Kavitha', 'Subjects': ['maths', 'physics', 'chemistry'], 1:[200, 198, 192]}
```

**PROGRAM 18**

**Simple Examples for Dictionary**

```
1. # Dictionary in Python
2.
3. # Different types of dictionaries
4. Nos_dict = {1: 'fruit', 2: 'vegetable', 3: 'fish'}
5. Hetro_dict = {'name': 'Kavitha', 'Subjects': ['maths',
'physics', 'chemistry'], 1: [200, 198, 192]}
6.
7. # Creating dictionary using dict function
8. D_dict = dict({'Name':'Renu','age':43})
9.
10. # Accessing using integer key
11. print Nos_dict[1]
12. # Accessing using string key
13. print Hetro_dict['Subjects']
14. print D_dict
```

**EXECUTION**

```
sh-4.3$ python program 18.py
fruit
['maths', 'physics', 'chemistry']
{'age': 43, 'Name': 'Renu'}
```

### 4.3.2 Deletion of Elements

The elements in the dictionary can be deleted by using del statement. The entire dictionary can be deleted by specifying the dictionary variable name alone as in line 14.

The general syntax of del statement is:

Syntax

```
for del
    del <dictionary_name>[key_value]
```

**PROGRAM 19**

**Simple Examples for Dictionary Deletion**

```
1. # Deletion of Dictionary
2.
3. dictForDel = {'name': 'Kavitha', 'Subjects': ['maths',
'physics', 'chemistry'], 1: [200, 198, 192]}
4.
5. # remove entry with key 'name'
6. del dictForDel['name']
7. print "dictForDel['Subjects']: ", dictForDel['Subjects']
8.
9. # remove all entries in dict
10. dictForDel.clear();
11. print "dictForDel : ", dictForDel
12.
13. # delete entire dictionary
14. del dictForDel
15. print "dictForDel : ", dictForDel
```

**EXECUTION**

```
sh-4.3$ python program 19.py
dictForDel['Subjects']: ['maths', 'physics', 'chemistry']
dictForDel : {}
dictForDel :
Traceback (most recent call last):
File "main.py", line 15, in <module>
print "dictForDel : ", dictForDel
NameError: name 'dictForDel' is not defined
```

### 4.3.3 Updating Elements

In dictionary the keys are always immutable. However the values are mutable. Hence only the values can be modified. The dictionary compound data type is mutable. New key-value pairs can be inserted or deletion from the dictionary.

**PROGRAM 20**

**Simple Examples for updation in Dictionary**

```
1. # Updates to Dictionary
2.
3. dictForUpd = {'name': 'Kavitha', 'Subjects': ['maths', 'physics', 'chemistry'], 1: [200, 198, 192]}
4.
5. # Modifying a value in dictionary
6. dictForUpd['name'] = 'Arjun'
7. print "dictForUpd: ", dictForUpd
8.
9. # Modifying the dictionary itself
10. dictForUpd['Age'] = 19
11. print "dictForUpd: ", dictForUpd
```

**EXECUTION**

```
sh-4.3$ python program 20.py
dictForUpd: {1: [200, 198, 192], 'Subjects': ['maths', 'physics', 'chemistry'], 'name': 'Arjun'}
dictForUpd: {1: [200, 198, 192], 'Age': 19, 'Subjects': ['maths', 'physics', 'chemistry'], 'name': 'Arjun'}
```

### 4.3.4 Dictionary Methods

Methods that are available for the dictionary are tabulated below. The following table gives the result of every method for the considered dictionary. In general the methods are accessed as DictionayVariable.MethodName()

| S.No. | Method | Description |
|---|---|---|
| 1. | clear() | Removes all elements of dictionary |
| 2. | copy() | Returns a shallow copy (alias) of the dictionary |
| 3. | fromkeys(seq, values) | Create a new dictionary with keys from seq and values |
| 4. | get(K, default=None) | For key K, returns value or default if key not in dictionary |
| 5. | has_key(K) | Returns true if key K is present; else false |
| 6. | items() | Returns a list of (key, value) tuple pairs |
| 7. | keys() | Returns list of dictionary keys |
| 8. | setdefault(K, default=None) | Set value for key K to default if key is not already |
| 9. | update(D) | Adds dictionary D's key-values pairs |
| 10. | values() | Returns list of values |
| 11. | pop(K[,d]) | Remove element with key K and return its value. Returns d if K is not found. |
| 12. | popitem() | Remove and return element( key, value) from the last |

Table 4.7 Methods in Dictionary

Considering Hetro_dict = {'name': 'Kavitha', 'Subjects': ['maths', 'physics', 'chemistry'], 1: [200, 198, 192]}

| S.No. | Method | Output |
|---|---|---|
| 1. | Temp= Hetro_dict.copy() | Temp={'name': 'Kavitha', 'Subjects': ['maths', 'physics', 'chemistry'], 1: [200, 198, 192]} |
| 2. | Hetro_dict.items() | [('name', 'Kavitha'), ('Subjects', ['maths', 'physics', 'chemistry'] ) , (1, [200, 198, 192] ) |
| 3. | Hetro_dict.keys() | [1, 'Subjects', 'name'] |
| 4. | Hetro_dict.values() | [[200, 198, 192], ['maths', 'physics', 'chemistry'], 'Kavitha'] |
| 5. | Hetro_dict.has_key('name')<br>Hetro_dict.has_key(3) | True<br>False |
| 6. | Hetro_dict.get(1,'NA')<br>Hetro_dict.get('Age','NA') | [200, 198, 192]<br>NA |
| 7. | Hetro_dict.setdefault('Age', 45)<br><br><br>Hetro_dict.setdefault(1, 100) | {1: [200, 198, 192], 'Age': 45, 'Subjects': ['maths', 'physics', 'chemistry'], 'name': 'Kavitha'}<br>No Change |
| 8. | Hetro_dict.pop('Sex', 'NA')<br>Hetro_dict.pop('name', 'NA') | NA<br>Kavitha |
| 9. | Hetro_dict.popitem() | ('name', 'Kavitha') |
| 10. | Hetro_dict.fromkeys (('Age', 'DOB'),10) | Hetro_dict = {'DOB': 10, 'Age': 10} |
| 11. | Dict1={'name': 'Kavitha'}<br>Hetro_dict.update(Dict1) | Hetro_dict = {'DOB': 10, 'Age': 10, 'name': 'Kavitha'} |
| 12. | Hetro_dict.clear() | Hetro_dict={} |

Table 4.8 Illustration of Methods in Dictionary

### 4.3.5 Dictionary Functions

The Tables 4.9 and 4.10 explain in detail the functions applicable for dictionary.

| S.No. | Functions | Description |
|---|---|---|
| 1. | cmp(dict1, dict2) | Compares elements in 2 dictionaries |
| 2. | len(dict) | Gives the number of elements in the dictionary. |
| 3. | str(dict) | Produces a printable string representation of a dictionary |
| 4. | type(variable) | Returns the type of the variable passed |
| 5. | del dict(K) | Deletes the element with key K from dict dictionary |

Table 4.9 Functions in Dictionary

| S.No. | Input | Functions | Output |
|---|---|---|---|
| 1. | dict1 = {'Name': 'Renu', 'Age': 7, 'Sex':'F'}<br>dict2 = {'Name': 'Mahnaz', 'Age': 27}<br>dict3 = {'Name': 'Renu', 'Age': 27}<br>dict4 = {'Name': 'Renu', 'Age': 27} | cmp(dict1, dict2)<br>cmp(dict2, dict3)<br>cmp(dict3, dict4) | 1<br>– 1<br>0 |
| 2. | dict4 = {'Name': 'Renu', 'Age': 27}<br>dict1 = {'Name': 'Renu', 'Age': 7, 'Sex':'F'} | len(dict4)<br>len(dict1) | 2<br>3 |
| 3. | dict1 = {'Name': 'Renu', 'Age': 7, 'Sex':'F'} | str(dict1) | {'Age': 7, 'Name': 'Renu', 'Sex': 'F'} |
| 4. | | type(dict1) | <type 'dict'> |
| 5. | | del dict1('Sex') | {'Name': 'Renu', 'Age': 7} |

Table 4.10 Illustration of Functions in Dictionary

## 4.4 ADVANCED LIST PROCESSING

### 4.4.1 List Comprehension

The concept of "list comprehensions" is used to construct lists in an easy and natural way. List comprehensions can also create a list with a sub-set of elements from another list by applying a condition. The list comprehensions make coding simpler and efficient; executes much faster than FOR loop. The below figure explains how a three lines code of FOR loop can be achieved by a single line of code using list comprehension. Program 21 illustrates the same.

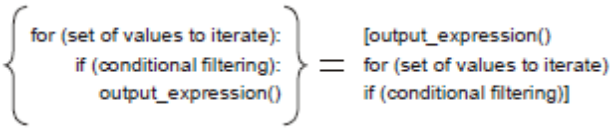


Figure 4.5 List Comprehension is equivalent to FOR loop

Here the output_expression is some calculation or operation acting upon the variable name. For each value in the list, the list comprehension; calculates a new value using expression. For each element of list, checks if it satisfies the conditional filtering condition. If the filter condition returns False, that element is omitted from the list before the list comprehension is evaluated. At last these new values are collected into a list which is the return value of the list comprehension. If list contains elements of different types, then expression must operate correctly on the types of all of list members. The output_expression can be an user-defined function too.

Examples

```
# Without Condition (A List with 0 to 9)
    x = [i for i in range (10)] => [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
Equivalent FOR
    X=[]
    for i in range(10):
        X.append(i)
    print X
# With Condition (A list with even nos in between 0-20)
    EvenNos = [n for n in range (20) if n % 2 == 0] => [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

Equivalent FOR

```
X= []
for i in range(20):
        if i % 2 == 0:
                X.append(i)
print X
```

**PROGRAM 21**

**Example programs for List Comprehension and List Comparison**

```
1. # List Comprehension & For Loop Comparison
2.
3. ##### WITHOUT CONDITION ########
4. List_Celsius = [30, 29, 43, -3]
5. List_Fahrenheit = [ ((float(9)/5)*x + 32) for x in List_Celsius]
6. print "List Comprehension :", List_Fahrenheit
7.
8. L_Fah= []
9. for i in List_Celsius:
10. L_Fah.append(((float(9)/5)*i + 32))
11. print "For Loop :",L_Fah
12.
13. ###### WITH CONDITION: Words with length greater than3########
14. List_words = ['the','bright', 'a','prettiest', 'an', 'three']
15. List_Len = [word for word in List_words if len(word)>3]
16. print "List Comprehension :", List_Len
17.
18. T= []
19. for i in List_words:
20. if len(i)>3 :
21. T.append(i)
22. print "For Loop :",T
```

**EXECUTION**

```
sh-4.3$ python program 21.py
List Comprehension : [86.0, 84.2, 109.4, 26.6]
For Loop : [86.0, 84.2, 109.4, 26.6]
List Comprehension : ['bright', 'prettiest', 'three']
For Loop : ['bright', 'prettiest', 'three']
```

**PROGRAM 22**

```
1. # List Comprehension
2.
3. from math import sqrt
```

```
4. words = 'I Love India'.split()
5. Cases_Len = [[w.upper(), w.lower(), len(w)] for w in words]
6. print Cases_Len
7.
8. # Nos divisible by 3 and 5 in first 50 nos
9. No_list = [x for x in range(1,50) if x % 3 == 0 if x % 5 == 0]
10. print "Divisible by 3 and 5 : ", No_list
11.
12. Pythagorus_list = [(x,y,z) for x in range(1,20) for y in range(x,20) for z in range(y,20) if x**2 +
y**2 == z**2]
13. print "Pythagorean Nos : " , Pythagorus_list
14.
15. #Non-Prime nos in the range of 1 to 20
16. n = 20
17. sqrt_n = int(sqrt(n))
18. no_primes = [j for i in range(2,sqrt_n) for j in range(i*2, n, i)]
19. print no_primes
20.
21. #Set of nos divisible by 2 and 3
22. print [(x,y) for x in range(1,7) if x % 2 == 0 for y in range(1,7)if y % 3 == 0]
```

**EXECUTION**

```
sh-4.3$ python program 22.py
[['I', 'i', 1], ['LOVE', 'love', 4], ['INDIA', 'india', 5]]
Divisible by 3 and 5 : [15, 30, 45]
Pythagorean Nos: [(3, 4, 5), (5, 12, 13), (6, 8, 10), (8, 15, 17), (9, 12, 15)]
Non Prime Nos : [4, 6, 8, 10, 12, 14, 16, 18, 6, 9, 12, 15, 18]
[(2, 3), (2, 6), (4, 3), (4, 6), (6, 3), (6, 6)]
```

**PROGRAM 23**

```
1. # List Comprehensions on dictionary and strings
2.
3. def sub(a,b) :
4. return (a-b)
5.
6. # Create a dictionary whose values are sorted lists
7. num = {'n1': [2, 3, 1], 'n2': [5, 1, 2], 'n3': [3, 2, 4]}
8. sorted_dict = {x: sorted(y) for x, y in num.items()}
9. print(sorted_dict)
10.
11. # Remove the elements whose key values lie in a range
12. dict1 = {0: [2, 3, 1], 2: [5, 1, 2], 5: [3, 2, 4]}
13. mydict = { k:v for k,v in dict1.items() if (k>1 and k<3) }
```

```
14. print(mydict)
15.
16. # Expression as user_defined function
17. Tlist = [(6, 3), (1, 7), (5, 5)]
18. print [sub(y, x) for (x, y) in Tlist]
19.
20. #Split and join can be used in a list comprehension
21. print " ".join( [s.capitalize() for s in "this is a test ".split( )] )
```

**EXECUTION**

```
sh-4.3$ python program 23.py
{'n1': [1, 2, 3], 'n2': [1, 2, 5], 'n3': [2, 3, 4]}
{2: [5, 1, 2]}
[-3, 6, 0]
This Is A Test
```

## 4.5 ILLUSTRATIVE PROGRAMS

### 4.5.1 Bubble Sort

Algorithm: Bubble Sort

Step 1: Start from the first element

Step 2: Check two adjacent elements in the list

Step 3: If second element is greater swap the positions

Step 4: Repeat this for entire list

Step 5: Repeat this until no swaps are needed

In bubble sort the list is passed through multiple times. During every pass adjacent items are compared and exchanged if they are out of order. During each pass the algorithm places the next largest value in the proper place. Each item "bubbles" up to the appropriate position in the list. Each pass is called iteration. In the below figure, for each iteration, the shaded items are the ones that are swapped. If the length of list is n, an element is compared with rest of n-1items. The figure 4.6 illustrates the bubble sort for list of nos given in iteration 1.

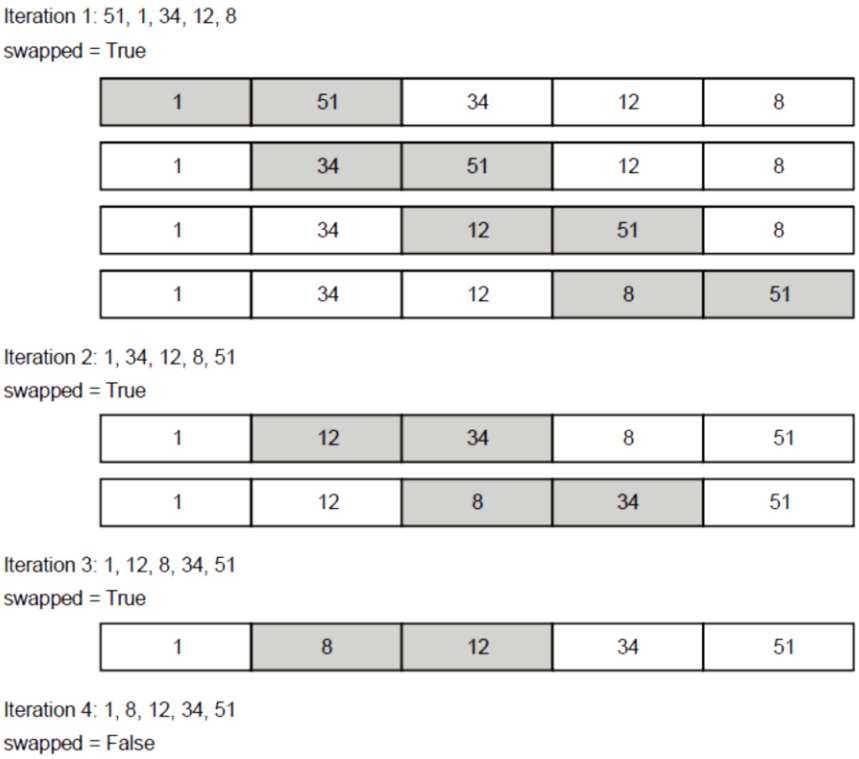


Figure 4.6 Bubble Sort of List

**PROGRAM 24**

**Python program for Bubble Sort**

```
1. # Bubble Sort
2. Nos_list = [54,26,93,17,77,31,44,55,20]
3. print 'Original List : ', Nos_list
4.
5. #Initially no nos are swapped
6. swapped = False
7.
8. while (swapped == False) :
9. # Pass through entire list
10. for i in range(0, len(Nos_list)-1):
11. # Check the adjacent nos
12. if Nos_list[i]>Nos_list[i+1]:
13. temp = Nos_list[i]
14. Nos_list[i] = Nos_list[i+1]
15. Nos_list[i+1] = temp
16. swapped = True
17. #If no swapping has happened break the while loop
18. if swapped == False:
19. break
```

```
20. #Swapping has happened; Continue to chech if further sorting is needed
21. swapped = False
22.
23. print 'Sorted List : ', Nos_list
```

**EXECUTION**

```
sh-4.3$ python program 24.py
Original List : [54, 26, 93, 17, 77, 31, 44, 55, 20]
Sorted List : [17, 20, 26, 31, 44, 54, 55, 77, 93]
```

### 4.5.2 Selection Sort

Algorithm: Selection Sort
Step 1: Assume element in index 0 as MIN
Step 2: Compare MIN will all the other elements in the list
Step 3: If an element lesser than MIN exits; place it in index 0
Step 4: Now assume element in index 1 as MIN; repeat steps 2 & 3
Step 5: Repeat until list is sorted

Selection sort is an improvement of bubble sort. On every pass on the list only one swap is done. Each time the largest element is pushed to the appropriate index. After the first iteration, the largest item is placed in the correct index. After the second iteration, the second largest is placed in correct index. This process continues until the entire list is sorted. For a list with n elements (n-1)*(n-1) iterations are required to sort.

Figure 4.7 shows the entire sorting process. The list is sorted in descending order in this program. Changing the "<" symbol to ">" will give the list in ascending order. As the lowest element is pushed to end after the iteration, the no of elements taken up for sorting in the next iteration is reduced by 1. In the below figure the smallest element is shaded in black and the elements shaded in grey are the ones that are swapped. In an iteration the lowest is moved to its appropriate index and that element is never again considered for comparison.

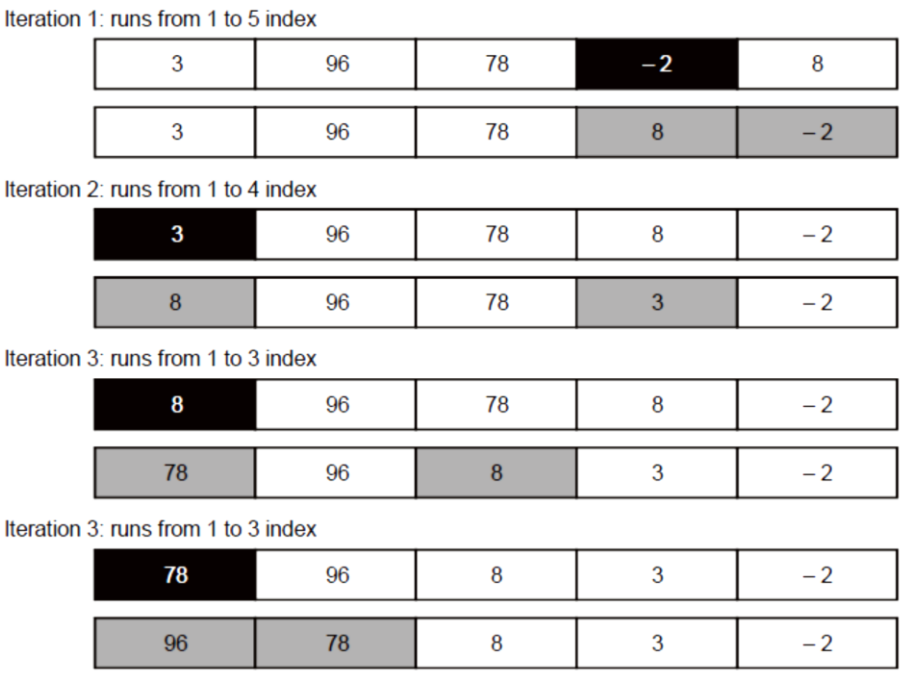


Figure 4.7 Selection Sort of List

**PROGRAM 25**

**Python program for Selection Sort**

```
1. # Selection Sort in Python
2.
3. # Function to swap two nos
4. def swap( Tlist, ind1, ind2 ):
5. tmp = Tlist[ind1]
6. Tlist[ind1] = Tlist[ind2]
7. Tlist[ind2] = tmp
8.
9. #List to be sorted
10. Nos_List = [7,2,5,1,29,6,4,19,11]
11. print "Before Sorting : ",Nos_List
12.
13. # For i=8, 7, 6, 5, 4, 3, 2, 1
14. for i in range(len(Nos_List)-1,0,-1):
15. MinPosition = 0
16. # Iteration specified in the figure
17. for k in range( 1 , i+1 ):
18. if Nos_List[k] < Nos_List[MinPosition]:
19. MinPosition = k
20. swap(Nos_List, MinPosition, i)
21.
```

22. print "After Sorting : ",Nos_List

**EXECUTION**

sh-4.3$ python program 25.py
Before Sorting : [7,2,5,1,29,6,4,19,11]
After Sorting : [29, 19, 11, 7, 6, 5, 4, 2, 1]

### 4.5.3 Insertion Sort

Algorithm: Insertion Sort
Step 1: Compare the first and second element; arrange in order
Step 2: Pick third element; Compare with elements before; arrange in order
Step 3: Pick next element; Compare with all elements before; arrange in order
Step 4: Repeat until till the last element; until the list is sorted

The Insertion Sort algorithm divides the input list into two parts: the sublist that is sorted, which is maintained in the left end of the list, and the sub-list that is yet to be sorted, which occupy the rest of the list. Initially, the sorted sublist is empty and the unsorted sub-list is the entire input list. The algorithm starts with the assumption that the sorted sub-list with first item alone is already sorted.

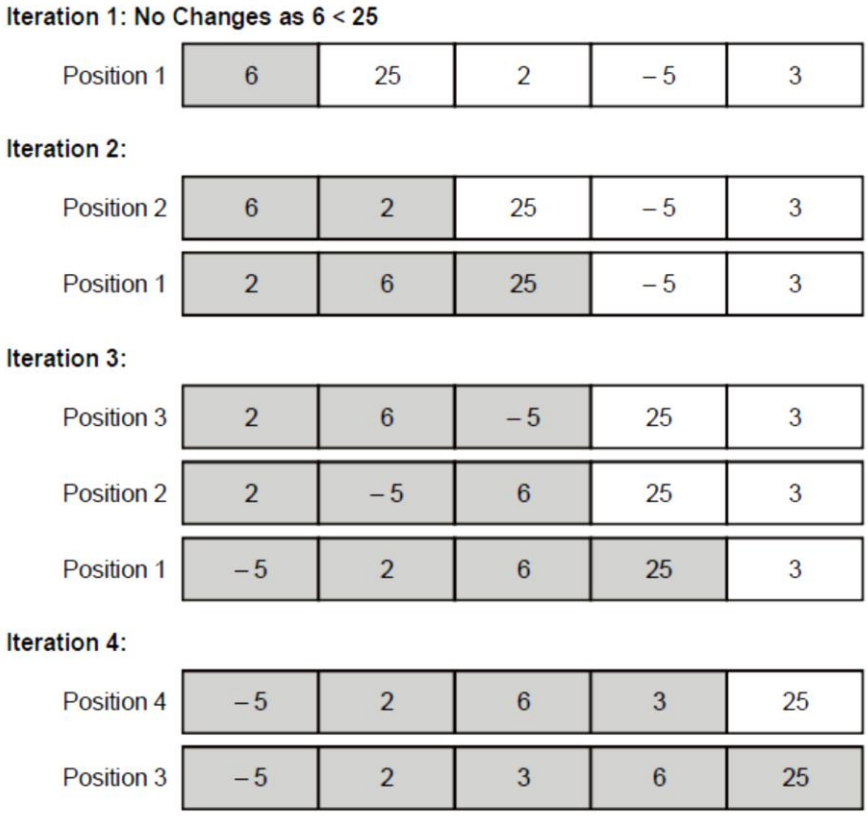


Figure 4.8 Insertion Sort of List

The smallest (or largest, in case of descending order) element in the unsorted sub-list is moved to leftmost position in the unsorted sub-list and that becomes part of sorted sub-list. The Insertion sort

doesn't use the swapping as performed in the previous algorithms but just shifts the elements. Selection sort is efficient on mutable data types like list.
The elements shaded in grey are the elements in sorted sub-list. The rest are the elements in unsorted sub-list. After the iteration, the sorted sub-list keeps growing. Each time the least element is passed on to the sorted sub-list and that sub-list is maintained in the correct order by the while loop in line 13.

**PROGRAM 26**
**Python program for Insertion Sort**

```
1. # Insertion Sort in Python
2.
3. Nos_List = [54,26,93,17,77,31,44,55,20]
4. print "Before Sorting: ", Nos_List
5.
6. # for i =1 to 9
7. for i in range( 1, len(Nos_List) ):
8. tmp = Nos_List[i]
9. # starts with second element
10. Position = i
11.
12. # This while loops creates the sorted sublist
13. while Position > 0 and tmp < Nos_List[Position - 1]:
14. # Compares with all elements before Position
15. Nos_List[Position] = Nos_List[Position - 1]
16. #Move Backwards and sort this sublist
17. Position -= 1
18.
19. # Move least to the least position
20. Nos_List[Position] = tmp
21.
22. print "After Sorting : ",Nos_List
```

**EXECUTION**

```
sh-4.3$ python program 26.py
Before Sorting : [54,26,93,17,77,31,44,55,20]
After Sorting : [17, 20, 26, 31, 44, 54, 55, 77, 93]
```

### 4.5.4 Merge Sort

Algorithm: Merge Sort
Step 1: If noofItems >= l return else continue
Step 2: Divide the list into two lists; firstList and secondList
Step 3: Sort firstList and secondList; by dividing each list further into halves
Step 4: Merge the sorted lists to get the sorted list.

Merge Sort uses divide and conquer strategy. It is a recursive algorithm that continually splits a list into half. A list that has more than one item is split into two halves and recursively the merge sort is invoked on both halves.

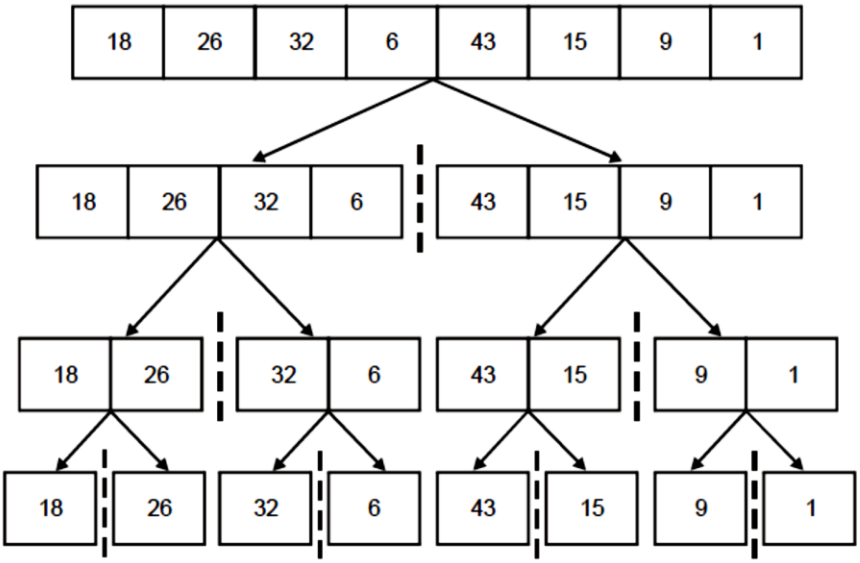


Figure 4.9 Splitting of List

Then the two halves are sorted and merged together. Two sorted lists are merged into a single sorted list. The figure 4.9 gives the illustration of how the lists are split into halves. The figure 4.10 gives the sorted list created by merging the sorted lists.

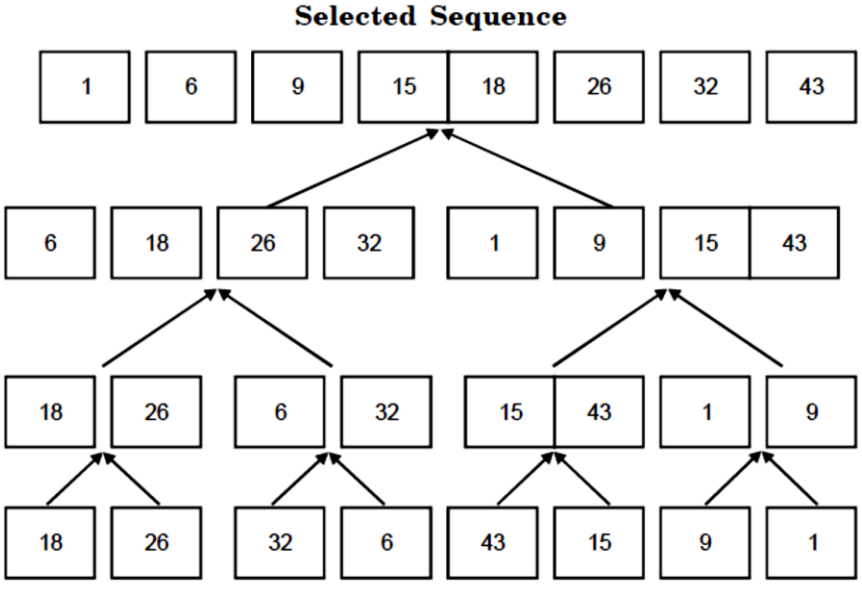


Figure 4.10 Merging of Sorted Lists

**PROGRAM 27**

**Python program for Merge Sort**

1. # Merge Sort in Python

```
2.
3. def mergesort( TList, first, last ):
4. # Divide the list into smaller lists
5. mid = ( first + last ) / 2
6. if first < last:
7. mergesort( TList, first, mid)
8. mergesort( TList, mid + 1, last)
9. # merge sorted lists into one
10. fIndex, lIndex = first, mid + 1
11. finalList = []
12. #Add to finalList smaller nos
13. while fIndex <= mid and lIndex <= last:
14. if TList[fIndex] < TList[lIndex] :
15. finalList.append(TList[fIndex])
16. fIndex += 1
17. else:
18. finalList.append(TList[lIndex])
19. lIndex += 1
20. # End of While; add the remaining sorted elements
21. if fIndex <= mid :
22. finalList = finalList + TList[fIndex:mid + 1]
23. if lIndex <= last:
24. finalList= finalList + TList[lIndex:last + 1]
25. #Add to TList all sorted nos
26. a=0
27. while first <= last:
28. TList[first] = finalList[a]
29. first += 1
30. a += 1
31.
32. #Main Program
33. Nos_List = [54,26,93,17,77,31,44,55,20]
34. print "Before Sorting : ",Nos_List
35. mergesort(Nos_List, 0, len( Nos_List ) - 1 )
36. print "After Sorting : ",Nos_List
```

**EXECUTION**

```
sh-4.3$ python program 27.py
Before Sorting : [54,26,93,17,77,31,44,55,20]
After Sorting : [17, 20, 26, 31, 44, 54, 55, 77, 93]
```

**4.5.5 Quick Sort**

Algorithm: Quick Sort

Step 1: Pick an element and assume it as PIVOT
Step 2: Partition
Arrange elements with values less than the PIVOT before it
Arrange elements with values greater than the PIVOT after it
Step 3: Recursively apply the above steps; for
List of elements with smaller values and greater values separately

The quick sort also uses divide and conquer technique. The algorithm selects the first element as the pivot value.

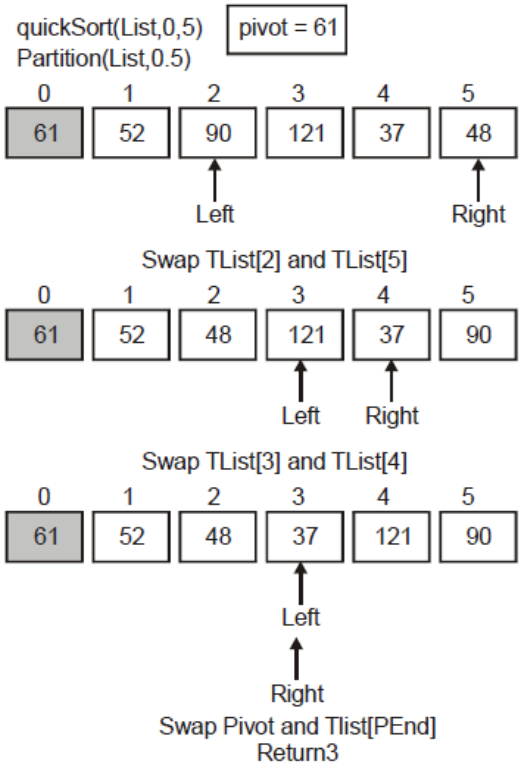


Figure 4.11 Quicksort Iteration 1

The pivot value plays the major role in splitting the list. The index of the pivot value in the final sorted list is called the split point. The split point will be used to divide the list for subsequent calls to the quick sort. The bold and shaded element is the pivot element. The list gets partitioned according to this pivot element.

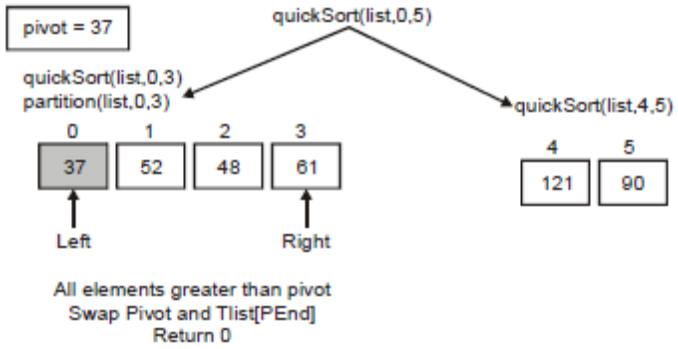


Figure 4.12 Quicksort after 1st Partition

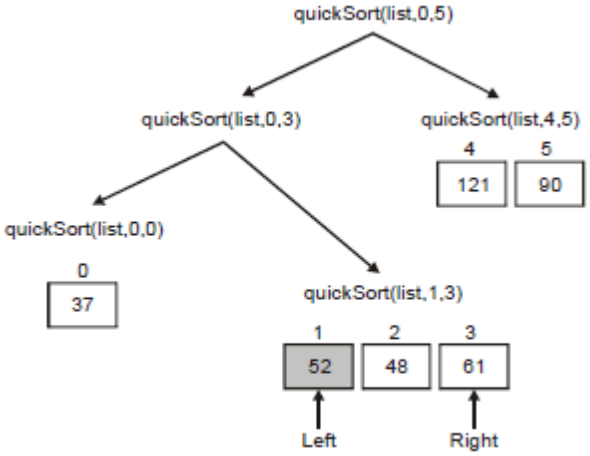


Figure 4.13 Quicksort after 2nd Partition

**PROGRAM 28**

**Python program for Quick Sort**

```
1. # Quick Sort in Python
2. def partition(TList, start, end):
3. pivot = TList[start]
4. # Start and End of partition area
5. PStart = start + 1
6. PEnd = end
7. done = False
8. while not done:
9. # Move further till you find a smaller element
10. while PStart <= PEnd and TList[PStart] <= pivot:
11. PStart = PStart + 1
12. # Move back till you find a greater element
13. while TList[PEnd] >= pivot and PEnd >=PStart:
14. PEnd = PEnd - 1
15. if PEnd < PStart:
```

```
16. done= True
17. else:
18. # swap places
19. temp=TList[PStart]
20. TList[PStart]=TList[PEnd]
21. TList[PEnd]=temp
22. # swap start with TList[PEnd]
23. temp=TList[start]
24. TList[start]=TList[PEnd]
25. TList[PEnd]=temp
26. return PEnd
27.
28. def quicksort(TList, start, end):
29. if start < end:
30. # partition the list
31. partInd = partition(TList, start, end)
32. # sort both halves
33. quicksort(TList, start, partInd-1)
34. quicksort(TList, partInd+1, end)
35. return TList
36.
37. Nos_List = [7,2,5,1,-29,6,4,-19,11]
38. print "Before Sorting : ",Nos_List
39. sortedList = quicksort(Nos_List,0,len(Nos_List)-1)
40. print "After Sorting : ",sortedList
```

**EXECUTION**

```
sh-4.3$ python program 28.py
Before Sorting : [7, 2, 5, 1, -29, 6, 4, -19, 11]
After Sorting : [-29, -19, 1, 2, 4, 5, 6, 7, 11]
```

### 4.5.6 Histogram

In normal English, histogram is a diagram consisting of rectangles whose area is proportional to the frequency of a variable and whose width is equal to the class interval.

A histogram is an accurate graphical representation of the distribution of numerical data. It is an estimate of the probability distribution of a continuous variable as show in the below picture. The program 29 tries to create a simple histogram; given the value for each bar.

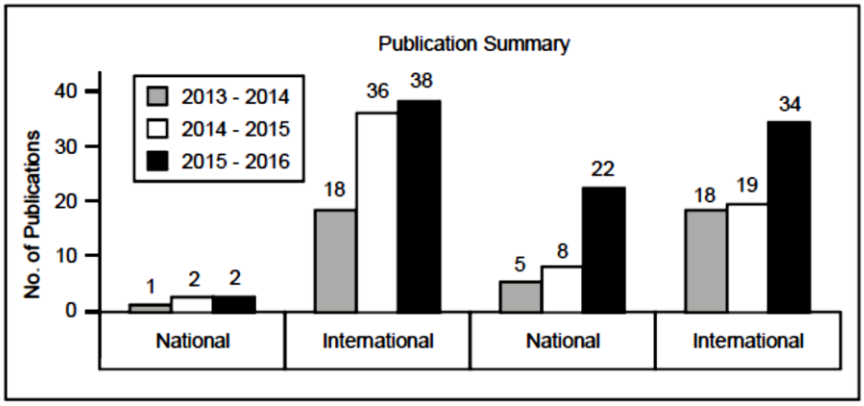


Figure 4.14 Graphical Histogram

**PROGRAM 29**

```
1. #Example for graphical histogram
2.
3. #Function to create histograms with = symbol
4. def histogram( items ):
5. #For every value print that many = symbol
6. for n in items:
7. output = ''
8. times = n
9. while( times > 0 ):
10. output += '='
11. times = times - 1
12. #Print the value after every output. That's the use of ,
13. print output,
14. print n
15.
16. #Call the function#
17. histogram([20, 33, 50, 15])
```

**EXECUTION**

```
sh-4.3$ python main.py
==================== 20
================================== 33
================================================== 50
=============== 15
```

In Python, when the data type like list, string or tuple are considered; by histogram we mean a function that counts the occurrence of element in a. The output of the function is a dictionary with element as key and count as value.

For example; Let L = 'abracadabra'

histogram (L) = {'a': 5, 'b': 2, 'c': 1, 'd': 1, 'r': 2}

**PROGRAM 30**

```
1. #Example for occurrencehistogram
2. #With counter method
3. from collections import Counter
4. print Counter("abracadabra")
5.
6. #Using List Comprehension
7. # For list data type
8. list_nos = [1,2,32,2,4,5,2,4]
9. print 'Using list and count method: ',
10. print {x: list_nos.count(x) for x in list_nos}
11.
12. # For tuple data type
13. tupleVal = ('hi', 'bye', 'python','book', 'bye','univ', 'bye','hi')
14. Dict_histogram= {}
15. print 'Using dictionary pop method: ',
16. for x in tupleVal: Dict_histogram[x]= Dict_histogram.pop(x,0) + 1
17. print Dict_histogram
18.
19. # For mixed list data type
20. L=[12,'pytho',(12,'check'),'pytho',(12,'check'), 'pytho']
21. Mix_Dict = {}
22. print 'Using dictionary setdefault method: ',
23. for x in L: Mix_Dict[x] = Mix_Dict.setdefault(x,0)+1
24. print Mix_Dict
```

**EXECUTION**

```
sh-4.3$ python program 30.py
Counter({'a': 5, 'r': 2, 'b': 2, 'c': 1, 'd': 1})
Using list and count method: {32: 1, 1: 1, 2: 3, 4: 2, 5: 1}
Using dictionary pop method: {'python':1, 'bye':3, 'hi': 2, 'univ':1, 'book': 1}
Using dictionary setdefault method:{(12, 'check'): 2, 12: 1, 'pytho': 3}
```

## TWO MARKS QUESTIONS WITH ANSWERS

1. Define List.

   A list is a data structure in Python that is a mutable, or changeable, ordered sequence of elements. Each element or value that is inside of a list is called an item. Each one of them is numbered, starting from zero - the first one is numbered zero, the second 1, the third 2, etc. You can remove values from the list, and add new values to the end

2. What is Cloning of List?

   Cloning is the process of modifying a list and also keeps a copy of the original. The entire list is copied not just the reference. The easiest way to clone a list is to use the slice operator.

   Example:

   a = [ 5, 10, 50, 100]

   b = a[0:2]

   b= [5, 10]

3. What is Aliasing?

   More than one list variable can point to the same data. This is called an Alias.

   For example:

   a = [ 5, 10, 50, 100]

   b = a

   Now, a and b both point to the same list.

   Without the slice operator, a and b point to the same list. Changes to one affect the other. With it, they point to different copies of the list that can be changed independently.

4. Define Tuple.

   A tuple is a sequence of immutable Python objects. Tuples are sequences, just like lists. The differences between tuples and lists are, the tuples cannot be changed unlike lists and tuples use parentheses, whereas lists use square brackets.

5. Explain Tuple Assignment with example.

   Python has a very powerful tuple assignment feature that allows a tuple of variables on the left of an assignment to be assigned values from a tuple on the right of the assignment.

   (name, surname, b_year, movie, m_year, profession, b_place) = julia

   This does the equivalent of seven assignment statements, all on one easy line. One requirement is that the number of variables on the left must match the number of elements in the tuple.

6. What is Slicing?

   A Python slice extracts elements, based on a start and stop. We specify an optional first index, an optional last index, and an optional step. For Example;

   ```
   values = [100, 200, 300, 400, 500]
   # Get elements from second index to Third index.
   slice = values[1:3]
   ```

```
# Slice from third index to index one from last.
slice = values[2:-1]
# Slice from start to second index.
slice = values[:2]
# Slice from second index to end.
slice = values[2:]
```

7. Define Dictionary.

Python dictionary is an unordered collection of items. While other compound data types have only value as an element, a dictionary has a key: value pair. Dictionaries are optimized to retrieve values when the key is known.

8. Give an example for List Comprehension

The concept of "list comprehensions" is used to construct lists in an easy and natural way. List comprehensions can also create a list with a sub-set of elements from another list by applying a condition. The list comprehensions make coding simpler and efficient; executes much faster than FOR loop.

For Example;

EvenNos = [n for n in range(20)if n % 2 == 0]=>[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]

9. What is Mutability?

A mutable object can be changed after it's created, and an immutable object can't. Lists are mutable and tuples are immutable.

10. List the functions of Tuple Data Type.

| S.No. | Function | Description |
|---|---|---|
| 1. | cmp(tuple1, tuple2) | Compares elements of both tuples. |
| 2. | len(tuple) | Gives the total length of the tuple. |
| 3. | max(tuple) | Returns item from the tuple with max value. |
| 4. | min(tuple) | Returns item from the tuple with min value. |
| 5. | tuple(seq) | Converts a sequence into tuple. |
| 6. | sum(tuple, S) | Return the sum of all elements and S |
| 7. | sorted(tuple) | Sorts the tuple in ascending or descending order |

11. List the methods of List Data Type.

| S. No. | Method | Description |
|---|---|---|
| 1. | append(E) | Add an element(E) to the end of the list |
| 2. | extend(seq) | Add all elements of a list to the another list(seq) |
| 3. | insert(ind,E) | Insert an element (E) at the defined index(ind) |
| 4. | remove(E) | Removes the first occurrence of element (E) from the list |
| 5. | pop(ind) | Removes and returns an element at the given index (ind) |
| 6. | index(E) | Returns the index of the first matched element with E |
| 7. | count(E) | Returns the count of number of elements (E) in the list |
| 8. | sort() | Sort items in a list in ascending order |
| 9. | reverse() | Reverse the order of items in the list |
| 10. | copy() | Returns a shallow copy of the list |

12. Comment on Tuple as Return type.

In general a functions can always only return a single value. However when the return datatype is a tuple we can combine together as many values as we like, and return them together.

For example, we could write a function that returns both the perimeter and the area of a square:

```
defArea_Perimeter(Side):
Area = Side * Side
Perimeter = 4 * Side
return (Perimeter, Area)
```

13. When a dictionary does is used instead of a list?

Dictionaries - are best suited when the data is labelled, i.e., the data is a record with field names.

Lists - are better option to store collections of un-labelled items say all the files and sub directories in a folder.

Generally Search operation on dictionary object is faster than searching a list object.

14. Differentiate between append() and extend() methods. ?

Both append() and extend() methods are the methods of list. These methods are used to add the elements at the end of the list.

append(element) - adds the given element at the end of the list which has called this method.

extend(another-list) - adds the elements of another-list at the end of the list which is called the extend method.

15. What is the output of print list + tinylist * 2 if list = [ 'abcd', 786 , 2.23, 'john', 70.2 ] and tinylist = [123, 'john']?

It will print concatenated lists. Output would be ['abcd', 786, 2.23, 'john', 70.2, 123, 'john', 123, 'john'].

16. What is the difference between tuples and lists in Python?
    The main differences between lists and tuples are - Lists are enclosed in brackets ( [ ] ) and their elements and size can be changed, while tuples are enclosed in parentheses ( ( ) ) and cannot be updated. Tuples can be thought of as read-only lists.

17. What is the difference between del() and remove() methods of list?
    To remove a list element, you can use either the del statement if you know exactly which element(s) you are deleting or the remove() method if you do not know.

18. How to merge two dictionaries?
    The update method update() merges the keys and values of one dictionary into another, overwriting values of the same key
    Dict1 = {"house":453,"Pet":"Cat","Color":"Brown"}
    Dict2 = {"Age":2","Color":"white"}
    w.update(w1)
    print w
    {"house":453,"Pet":"Cat","Age":2","Color":"white"}

19. Define mutable and immutable data type.
    Immutable data value: A data value which cannot be modified. Assignments to elements or slices (sub-parts) of immutable values cause a runtime error. Mutable data value: A data value which can be modified. The types of all mutable values are compound types. Lists and dictionaries are mutable; strings and tuples are not.

20. When is a dictionary used instead of a list?
    Dictionaries - are best suited when the data is labelled, i.e., the data is a record with field names.
    Lists - are better option to store collections of un-labelled items say all the files and sub directories in a folder.
    Generally Search operation on dictionary object is faster than searching a list object.

**REVIEW QUESTIONS**

1. Illustrate with suitable examples the methods of Lists.
2. Explain in detail List data type.
3. Explain with examples Aliasing and Cloning in List.
4. Explain in detail List Slicing
5. Explain the data type dictionary.
6. Write a Python script to generate and print a dictionary that contains a number (between 1 and n) in the form (x, x*x).
7. What are tuples? Explain in detail the immutable property of tuple.
8. Write a Python program to find number of times each element is present in the tuple.

Example: tuplex = 2, 4, 5, 5, 2, 5, 4, 4, 5
Output:
Number 2 - 2 times
Number 4 - 3 times
Number 5 - 4 times

9. With example programs illustrate List Comprehensions.