LISTS, TUPLES, DICTIONARIES

4

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

- A **list** is a sequence or collection of values
- A list is a sequence like a string,
- In a string, the values are characters
- In a list, these values can be any data type.
- The values in a list are called **elements** or sometimes **items**.
- There are several ways to create a new list
- The simplest is to enclose the elements in square brackets ([ ]):

```
[10, 20, 30, 40]
['crunchy frog', 'ram bladder', 'lark vomit']
```

- The elements of a list don't have to be the same data type.
- The following list contains a string, a float, an integer, and (lo!) another list:

```
['spam', 2.0, 5, [10, 20]]
```

- A list within another list ['spam',2.0,5,[10,20]] is **nested**.
- A list that contains no elements is called an empty list
- we can create one with empty brackets, [].
- We can assign list values to variables

```
>>> cheeses = ['Cheddar', 'Edam', 'Gouda']
>>> numbers = [42, 123]
>>> empty = []
>>> print(cheeses, numbers, empty)
['Cheddar', 'Edam', 'Gouda'] [42, 123] []
```

4.1.1 List operations

- Various Operations can be performed on List.
- Operations performed on List are given as:

a) Adding Lists: (concatenating two list)

- Lists can be added by using the concatenation operator(+) to join two lists.

Example:

```
list1=[10,20]
   list2=[30,40]
   list3=list1+list2
   print list3
```

Output:

```
>>>
   [10, 20, 30, 40]
   >>>
```

- '+'operator implies that both the operands passed must be list else error will be shown.

Example:

```
 list1=[10,20]
list1+30
print list1
```

Output:

```
Traceback (most recent call last):
```

```
    File "C:/Python27/lis.py", line 2, in <module>
      list1+30
```

b) Replicating lists:

- Replicating means repeating.
- It can be performed by using '*' operator by a specific number of time.

Example:

```
list1=[10,20]
print list1*1
```

Output:

```
>>>
[10, 20]
>>>
```

c) List slicing:

- A subpart of a list can be retrieved on the basis of index.
- This subpart is known as list slice.

Example:

```
list1=[1,2,4,5,7]
print list1[0:2]
print list1[4]
list1[1]=9
print list1
```

Output:

```
>>>
[1, 2]
7
[1, 9, 4, 5, 7]
>>>
```

Other Operations:

- Apart from above operations various other functions can also be performed on List such as Updating, Appending and Deleting elements from a List:

a) Updating elements in a List:

- To update or change the value of particular index of a list, assign the value to that particular index of the List.

Syntax:

```
<list_name>[index]=<value>
```

Example:

```
data1=[5,10,15,20,25]
print "Values of list are: "
print data1
data1[2]="Multiple of 5"
print "Values of list are: "
print data1
```

Output:

```
>>>
Values of list are:
[5, 10, 15, 20, 25]
Values of list are:
[5, 10, 'Multiple of 5', 20, 25]
>>>
```

b) Appending elements to a List:

- append() method is used to append i.e., add an element at the end of the existing elements.

Syntax:

```
<list_name>.append(item)
```

Example:

```
list1=[10,"rahul",'z']
print "Elements of List are: "
print list1
list1.append(10.45)
print "List after appending: "
print list1
```

Output:

```
>>>
Elements of List are:
[10, 'rahul', 'z']
List after appending:
[10, 'rahul', 'z', 10.45]
>>>
```

c) Deleting Elements from a List:

- del statement can be used to delete an element from the list.
- It can also be used to delete all items from startIndex to endIndex.

Example:

```
list1=[10,'rahul',50.8,'a',20,30]
print list1
del list1[0]
print list1
del list1[0:3]
print list1
```

Output:

```
>>>
[10, 'rahul', 50.8, 'a', 20, 30]
```

```
['rahul', 50.8, 'a', 20, 30]
[20, 30]

>>>
```

- The + operator concatenates lists:

```
>>> a = [1, 2, 3]
>>> b = [4, 5, 6]
>>> c = a + b
>>> c
[1, 2, 3, 4, 5, 6]
```

- The * operator repeats a list a given number of times:

```
>>> [0] * 4
[0, 0, 0, 0]
>>> [1, 2, 3] * 3
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

- The first example repeats [0] four times.
- The second example repeats the list [1, 2, 3] three times.

d) Traversing a list or list looping

- Traversing a list means accessing the elements in the list
- For loop is mainly used for list traversal

Traversing a list of string

```
for friend in ['Margot', 'Kathryn', 'Prisila']:
    invitation = "Hi " + friend + ".  Please come to my party on Saturday!"
    print(invitation)
```

Output

```
Hi friend Margot
Hi friend Karthryn
Hi friend Prisila
```

4.1.2 List slices

- The slice operator also works on lists:

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> t[1:3]
['b', 'c']
>>> t[:4]
['a', 'b', 'c', 'd']
>>> t[3:]
['d', 'e', 'f']
```

- If you omit the first index, the slice starts at the beginning.
- If you omit the second, the slice goes to the end.
- So if you omit both, the slice is a copy of the whole list.

```
>>> t[:]
['a', 'b', 'c', 'd', 'e', 'f']
```

- Since lists are mutable, it is often useful to make a copy before performing operations that modify lists.
- A slice operator on the left side of an assignment can update multiple elements:

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> t[1:3] = ['x', 'y']
>>> t
['a', 'x', 'y', 'd', 'e', 'f']
```

4.1.3. List methods

- Python provides methods that operate on lists.
- For example, append adds a new element to the end of a list:

Syntax:

```
<list object>.<methodname>()
```

Or

<list name>.<methodname>(paramenters)

```
>>> t = ['a', 'b', 'c']
>>> t.append('d')
>>> t
['a', 'b', 'c', 'd']
```

- extend takes a list as an argument and appends all of the elements:

```
>>> t1 = ['a', 'b', 'c']
>>> t2 = ['d', 'e']
>>> t1.extend(t2)
>>> t1
['a', 'b', 'c', 'd', 'e']
```

- to sort arranges the elements of the list from low to high:

```
>>> t = ['d', 'c', 'e', 'b', 'a']
>>> t.sort()
>>> t
['a', 'b', 'c', 'd', 'e']
```

There are following built-in methods of List:

Methods	**Description**
index(object)	Returns the index value of the object.
count(object)	It returns the number of times an object is repeated in list.
pop()/pop(index)	Returns the last object or the specified indexed object. It removes the popped object.
insert(index,object)	Insert an object at the given index.
extend(sequence)	It adds the sequence to existing list.
remove(object)	It removes the object from the given List.
reverse()	Reverse the position of all the elements of a list.

sort()	It is used to sort the elements of the List.

1) index(object):

Example:

```
data = [786,'abc','a',123.5]
print "Index of 123.5:", data.index(123.5)
print "Index of a is", data.index('a')
```

Output:

```
>>>
Index of 123.5 : 3
Index of a is 2
>>>
```

2) count(object):

Example:

```
data = [786,'abc','a',123.5,786,'rahul','b',786]
print "Number of times 123.5 occured is", data.count(123.5)
print "Number of times 786 occured is", data.count(786)
```

Output:

```
>>>
Number of times 123.5 occured is 1
Number of times 786 occured is 3

>>>
```

3) pop()/pop(int):

Example:

```
data = [786,'abc','a',123.5,786]
print "Last element is", data.pop()
```

```
print "2nd position element:", data.pop(1)
print data
```

Output:

```
>>>
Last element is 786
2nd position element:abc
[786, 'a', 123.5]
>>>
```

4) insert(index,object):

Example:

```
data=['abc',123,10.5,'a']
data.insert(2,'hello')
print data
```

Output:

```
>>>
['abc', 123, 'hello', 10.5, 'a']
>>>
```

5) extend(sequence):

Example:

```
data1=['abc',123,10.5,'a']
data2=['ram',541]
data1.extend(data2)
print data1
```

```
print data2
```

Output:

```
>>>
['abc', 123, 10.5, 'a', 'ram', 541]
['ram', 541]
>>>
```

6) remove(object):

Example:

```
data1=['abc',123,10.5,'a','xyz']
data2=['ram',541]
print data1
data1.remove('xyz')
print data1
print data2
data2.remove('ram')
print data2
```

Output:

```
>>>
['abc', 123, 10.5, 'a', 'xyz']
['abc', 123, 10.5, 'a']
['ram', 541]
[541]
>>>
```

7) reverse():

Example:

```
list1=[10,20,30,40,50]
list1.reverse()
```

```
print list1
```

Output:

```
>>>
[50, 40, 30, 20, 10]
>>>
```

8) sort():

Example:

```
list1=[10,50,13,'rahul','aakash']
list1.sort()
print list1
```

Output:

```
>>>
[10, 13, 50, 'aakash', 'rahul']

>>>
```

4.1.4 List loop (Traversing a list)

- The most common way to traverse the elements of a list is with a for loop.
- The syntax is the same as for strings:

```
for cheese in cheeses:
        print(cheese)
```

- To write or update the elements, you need the indices.
- A common way to do that is to combine the built-in functions range and len:

```
for i in range(len(numbers)):
        numbers[i] = numbers[i] * 2
```

```
```

- This loop traverses the list and updates each element.
- len returns the number of elements in the list.
- range returns a list of indices from 0 to n □ 1, where n is the length of the list.
- Each time through the loop i gets the index of the next element.
- The assignment statement in the body uses i to read the old value of the element and to assign the new value.

- A for loop over an empty list never runs the body:

```
for x in []:
        print('This never happens.')
```

- Although a list can contain another list, the nested list still counts as a single element.
- The length of this list is four:

```
['spam', 1, ['Brie', 'Roquefort', 'Pol le Veq'], [1, 2, 3]]
```

Traversing a list or list looping

- Traversing a list means accessing the elements in the list
- For loop is mainly used for list traversal

Traversing a list of string

```
for friend in ['Margot', 'Kathryn', 'Prisila']:
   invitation = "Hi " + friend + ".  Please come to my party on Saturday!"
   print(invitation)
```

Output

Hi friend Margot
Hi friend Karthryn
Hi friend Prisila

4.1.5 Mutability

- Unlike strings, lists are mutable.
- Mutable means the elements in the list can be changed.

Example:

```
>>> numbers = [42, 123]
>>> numbers[1] = 5
>>> numbers
[42, 5]
```

- The one-eth element of numbers, which used to be 123, is now 5.
- List indices work the same way as string indices:
    1. Any integer expression can be used as an index.
    2. If you try to read or write an element that does not exist, you get an IndexError.
    3. If an index has a negative value, it counts backward from the end of the list.

- The in operator also works on lists.

```
>>> cheeses = ['Cheddar', 'Edam', 'Gouda']
>>> 'Edam' in cheeses
True
>>> 'Brie' in cheeses
False
```

4.1.6 Aliasing

- If a refers to an object and you assign b = a, then both variables refer to the same object:

```
>>> a = [1, 2, 3]
>>> b = a
>>> b is a
True
```

- The association of a variable with an object is called a **reference**.
- In this example, there are two references to the same object.
- An object with more than one reference has more than one name, so we say that the object is **aliased**.
- If the aliased object is mutable, changes made with one alias affect the other.

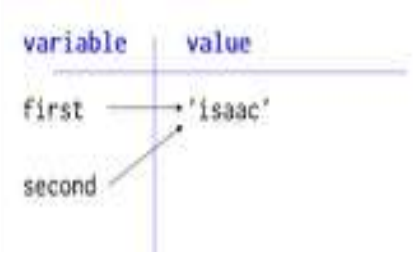
```
>>> b[0] = 42
>>> a
[42, 2, 3]
```

- An *alias* is a second name for a piece of data.
- Programmers create aliases because it's often easier (or more useful) to have a second way to refer to data than to copy it.
- If the data is immutable—i.e., if it cannot be modified in place—then aliasing doesn't matter
- because if the data can't change, it doesn't make a difference how many times it's referred to.
- But if data *can* change in place, then aliasing can lead to some hard-to-find bugs.
- In Python, aliasing happens whenever one variable's value is assigned to another variable, because variables are just names that store references to values.

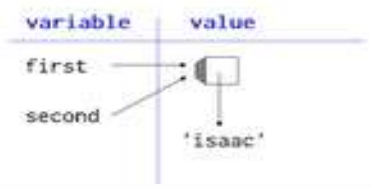
For example

```
First = Isaac

Second = first
```

- This copies the reference in first to second, after which second refers to the same string in memory as first

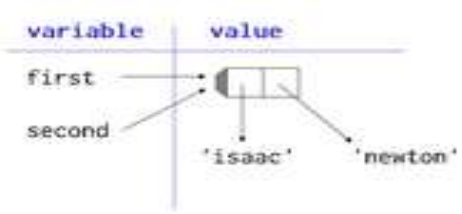


- However, this doesn't happen with lists: they can be changed in place.
- Let's assign a list containing the string ‘isaac’ to the variable first …



- and then assign first to second
- The two variables are now referring to the same thing in memory as before.
- If we now append another string to the list that first is pointing at…

```
First = first.append('newton')
```



- The change is also visible when we look at second's value, because it's the same value.
- We didn't explicitly modify second —there's nothing in the expression first.append('newton')to indicate that the value of second will change—but the change happens nonetheless.
- This is called a *side effect*,
- These side effects can lead to some hard-to-find bugs.
- So if aliasing can cause bugs, why does Python allow it?

1. Some languages don't Or at least appear not to
2. Aliasing a million element list is more efficient than copying it
3. Used when you really want to update the structure in place.

Example: Program for Aliasing a list

```
l1=[1,3,5]
l2=l1
print("the values of l1 are:", l1)
print("the values of l2 are:", l2)
l1[2]=4
print("the values of new l1 are:", l1)
print("the values of  new l2 are:", l2)
```

Output

```
the values of l1 are: [1,3,5]
the values of l2 are: [1,3,5]
the values of new l1 are: [1,3,4]
the values of  newl2 are: [1,3,4]
```

4.1.7 Cloning lists

- **Cloning** is the technique of making an identical copy of something.
- Cloning in list is explained by Author RP has explained in his blog http://tech.meetrp.com/blog/methods-to-clone-a-python-list/ as

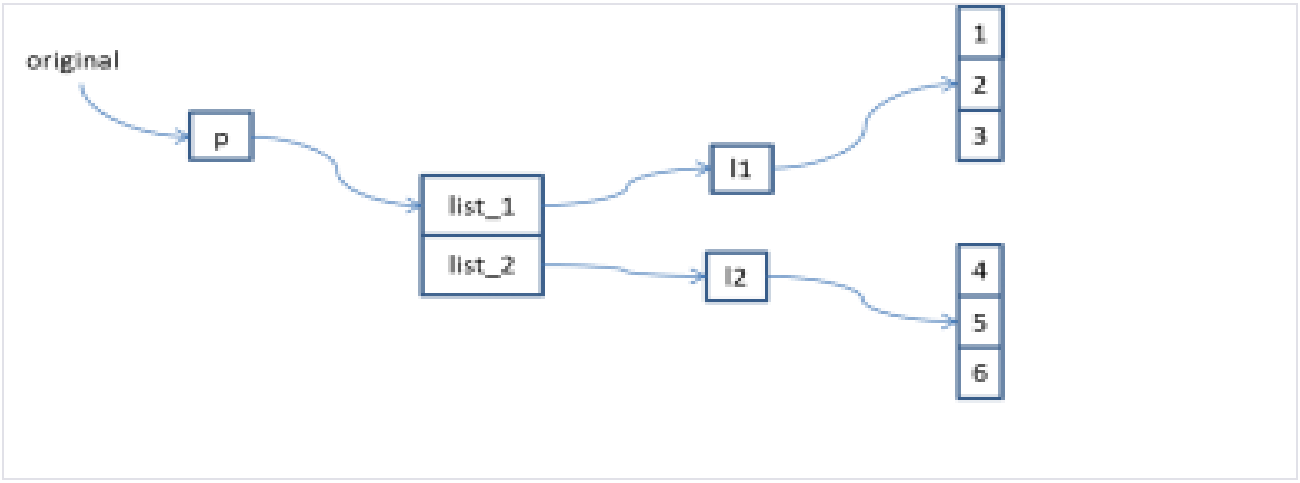
Methods to clone a python list

- Python list are one of the most used data structure.
- Basic understanding of how the list object is implemented within will give an edge on how to use the list esp from cloning perspective.
- Most important point in lists is it is mutable.
- A list object contains a list of pointers to pyobjects.
- A pyobject contains a pointer to the corresponding type object.

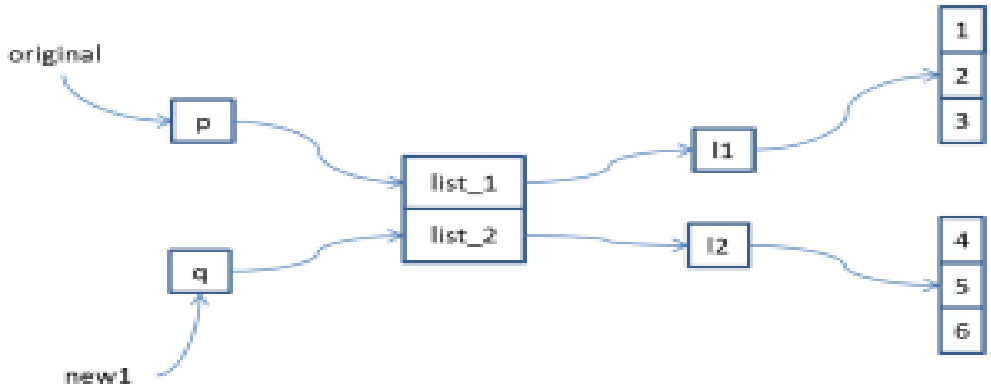
Clone by assignment

- The new list "new" points to the original list. so, any changes to the original list will be reflected in the new list
- clone by assignment results in a list that is tightly coupled with the original list. so any updates to the original list is seen on the cloned list.

```
1. LIST_1 = [1,2,3]
2. LIST_2 = [4,5,6]
3. ORIGINAL = [ LIST_1, LIST_2 ]
```




```
1. NEW = ORIGINAL
```



Example:

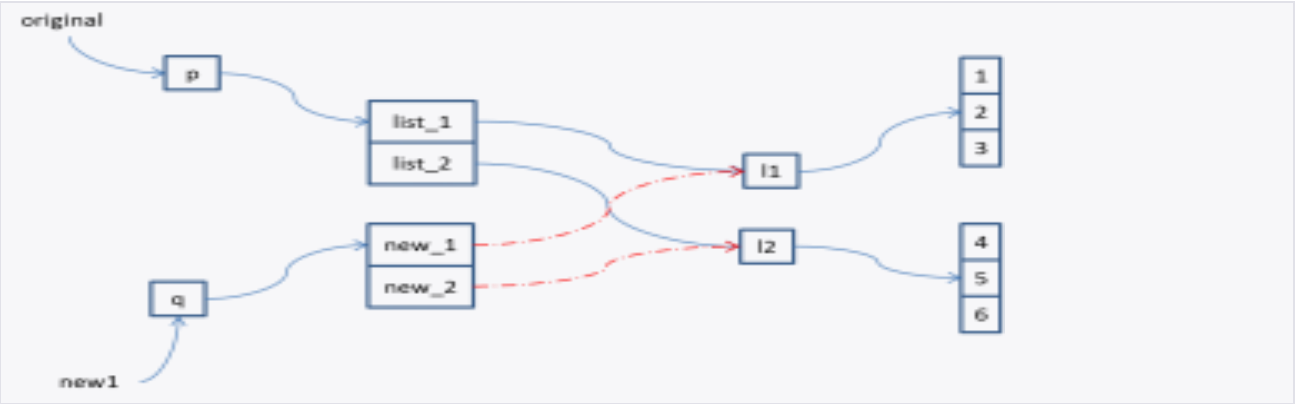
```
1. LIST_1 = [1,2,3]
2. LIST_2 = [4,5,6]
3. ORIGINAL = [ LIST_1, LIST_2 ]
4.
5. NEW = ORIGINAL
6.
7. PRINT 'ORIGINAL: ', ORIGINAL
8. PRINT 'NEW: ', NEW
9.
10.       LIST_2.APPEND(7)
11.       LIST_3 = [8, 9]
12.       ORIGINAL.APPEND(LIST_3)
13.
14.       PRINT '**********************************'
15.       PRINT 'ORIGINAL: ', ORIGINAL
16.       PRINT 'NEW: ', NEW1
```

```
$ PYTHON CLONE-BY-ASSIGN.PY

ORIGINAL:  [[1, 2, 3], [4, 5, 6]]

NEW:  [[1, 2, 3], [4, 5, 6]]

**********************************

ORIGINAL:  [[1, 2, 3], [4, 5, 6, 7], [8, 9]]

NEW:  [[1, 2, 3], [4, 5, 6, 7], [8, 9]]
```

Clone by slice

- The new list “new” creates a copy of original list but shares the content with the original list. so, any changes to the original list will **not** affect the new list but changes to the content will alter the new2 as well
- as it can be seen, clone by slice results in a list that is not tightly coupled with the original list but tightly coupled with the content of the original list. so, updates to the original list does not affect the cloned list but updates to the content of the original list affect the cloned list.

```
1. NEW = ORIGINAL[:]
```




```
1. LIST_1 = [1,2,3]
2. LIST_2 = [4,5,6]
3. ORIGINAL = [ LIST_1, LIST_2 ]
4.
5. NEW = ORIGINAL[:]
6.
7. PRINT 'ORIGINAL: ', ORIGINAL
8. PRINT 'NEW: ', NEW
9.
10.  LIST_2.APPEND(7)
11.  LIST_3 = [8, 9]
12.  ORIGINAL.APPEND(LIST_3)
13.
14.  PRINT '*********************************'
15.  PRINT 'ORIGINAL: ', ORIGINAL
16.  PRINT 'NEW: ', NEW
```

```
$ PYTHON CLONE-BY-SLICE.PY

ORIGINAL:  [[1, 2, 3], [4, 5, 6]]
```

```
NEW:  [[1, 2, 3], [4, 5, 6]]

**********

ORIGINAL:  [[1, 2, 3], [4, 5, 6, 7], [8, 9]]

NEW:  [[1, 2, 3], [4, 5, 6, 7]]
```

Clone by copy constructor

```
1. NEW = LIST(ORIGINAL)
```

- This is exactly same as the clone by slice.

```
$ PYTHON CLONE-BY-COPY-CONSTRUCTOR.PY

ORIGINAL:  [[1, 2, 3], [4, 5, 6]]

NEW:  [[1, 2, 3], [4, 5, 6]]

**********

ORIGINAL:  [[1, 2, 3], [4, 5, 6, 7], [8, 9]]

NEW:  [[1, 2, 3], [4, 5, 6, 7]]
```

Clone by copy module

- There is a copy module that can be imported and gives 2 different methods to copy:

    1. copy()
    2. deepcopy()

Cloning by copy()

- This is exactly same as slice & copy-constructor.

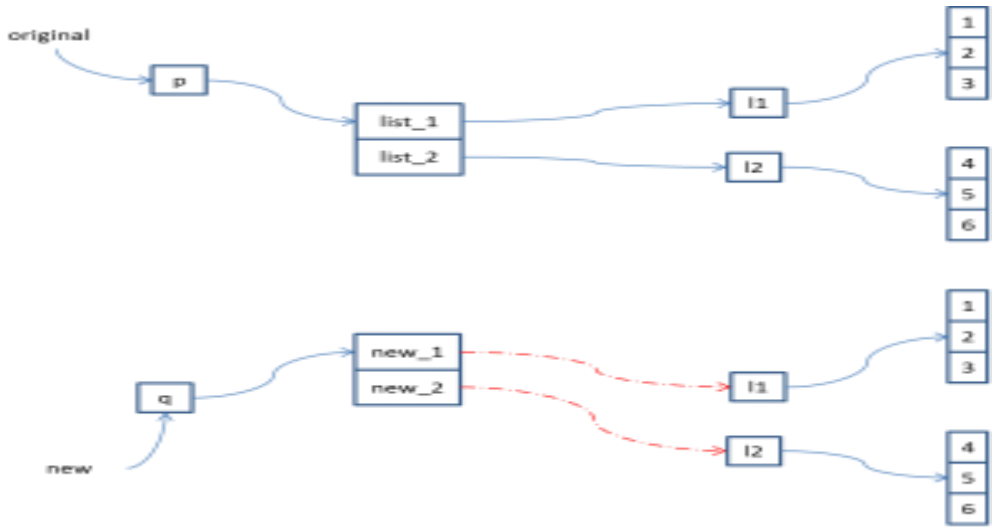
```
1. IMPORT COPY
2. NEW = COPY.COPY(ORIGINAL)
```

- This is exactly same as the clone by slice, returns a shallow copy of the original.

```
$ PYTHON CLONE-BY-COPY.PY

ORIGINAL:  [[1, 2, 3], [4, 5, 6]]

NEW:  [[1, 2, 3], [4, 5, 6]]

**********************************

ORIGINAL:  [[1, 2, 3], [4, 5, 6, 7], [8, 9]]

NEW:  [[1, 2, 3], [4, 5, 6, 7]]
```

Cloning by deepcopy()

- This is the ultimate copy that we might require.
- This creates an entirely new list, which has no bindings whatsoever with the original list.



- create clone using deepcopy function from the copy module

```
1. LIST_1 = [1,2,3]
2. LIST_2 = [4,5,6]
3. ORIGINAL = [ LIST_1, LIST_2 ]
4.
```

```
5. NEW = COPY.DEEPCOPY(ORIGINAL)
6.
7. PRINT 'ORIGINAL: ', ORIGINAL
8. PRINT 'NEW: ', NEW
9.
10.      LIST_2.APPEND(7)
11.      LIST_3 = [8, 9]
12.      ORIGINAL.APPEND(LIST_3)
13.
14.      PRINT '*********************************'
15.      PRINT 'ORIGINAL: ', ORIGINAL
16.      PRINT 'NEW: ', NEW
```

OUTPUT

```
$ PYTHON CLONE-BY-DEEPCOPY.PY

ORIGINAL:  [[1, 2, 3], [4, 5, 6]]

NEW:  [[1, 2, 3], [4, 5, 6]]

*********************************

ORIGINAL:  [[1, 2, 3], [4, 5, 6, 7], [8, 9]]

NEW:  [[1, 2, 3], [4, 5, 6]]
```

4.1.8 List parameters

- When you pass a list to a function, the function gets a reference to the list.
- If the function modifies the list, the caller sees the change.
- For example, delete_head removes the first element from a list:

Syntax

```
def delete_head(t):
del t[0]
```

Example 1:

```
>>> letters = ['a', 'b', 'c']
>>> delete_head(letters)
>>> letters
['b', 'c']
```

- The parameter t and the variable letters are aliases for the same object.

Example 2:

```
>>> letters = ['a', 'b', 'c']
>>> rest = tail(letters)
>>> rest
['b', 'c']
```

Passing a list as a parameter to function

- The elements of list will change when passing a list as parameter to a function.
- This is because the list is mutable

Example:

```
import math

def square(x):
    result = []
    for y in x:
        result.append(math.pow(y,2.0))
    return result

print(square([1,2,3]))
```

Output

```
[1.0, 4.0, 9.0]
```

Returning a list as output from a function

Example

```
import math

def square(x):
    return [math.pow(y, 2) for y in x]

>>> print(square([1,2,3]))
[1.0, 4.0, 9.0]
```

4.2 Tuples

- **Tuples are immutable**
- A tuple is a sequence of values. Therefore it cannot be changed
- The objects are enclosed within parenthesis and separated by comma.
- The values can be any type, and they are indexed by integers, so in that respect tuples are a lot like lists.
- Tuple is similar to list. Only the difference is that list is enclosed between square bracket, tuple between parenthesis and List have mutable objects whereas Tuple have immutable objects.

```
>>> data=(10,20,'ram',56.8)
>>> data2="a",10,20.9
>>> data
(10, 20, 'ram', 56.8)
>>> data2
('a', 10, 20.9)

>>>
```

- If Parenthesis is not given with a sequence , it is by default treated as tuple.
- There can be an empty Tuple also which contains no object.

Example:

```
tuple1=()
```

- For a single valued tuple, there must be a comma at the end of the value.

Example

```
tupl1='a','mahesh',10.56
   tupl2=tupl1,(10,20,30)
   print tupl1
   print tupl2

Output:


>>>
('a', 'mahesh', 10.56)
(('a', 'mahesh', 10.56), (10, 20, 30))
```

- Syntactically, a tuple is a comma-separated list of values:

```
>>> t = 'a', 'b', 'c', 'd', 'e'
```

- Although it is not necessary, it is common to enclose tuples in parentheses:

```
>>> t = ('a', 'b', 'c', 'd', 'e')
```

- To create a tuple with a **single element,** you have to include a final comma:

```
>>> t1 = 'a',
>>> type(t1)
<class 'tuple'>
```

- A value in parentheses is not a tuple:

```
>>> t2 = ('a')
>>> type(t2)
<class 'str'>
```

- Another way to create a tuple is the built-in function tuple. With no argument, it creates an empty tuple:

```
>>> t = tuple()
>>> t
()
```

- If the argument is a sequence (string, list or tuple), the result is a tuple with the elements of the sequence:

```
>>> t = tuple('lupins')
>>> t
('l', 'u', 'p', 'i', 'n', 's')
```

- Because tuple is the name of a built-in function, you should avoid using it as a variable name.

Accessing Tuple

- Tuple can be accessed in the same way as List.

Example

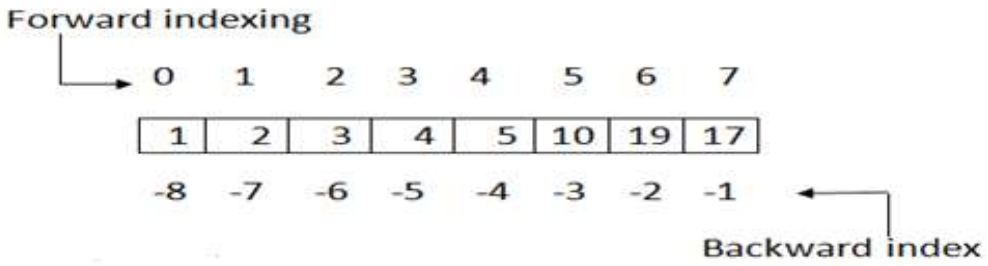
```
data1=(1,2,3,4)
data2=('x','y','z')
print data1[0]
print data1[0:2]
print data2[-3:-1]
print data1[0:]
print data2[:2]
```

Output:

```
>>>
1
(1, 2)
('x', 'y')
(1, 2, 3, 4)
('x', 'y')
>>>
```

Elements in a Tuple

```
Data=(1,2,3,4,5,10,19,17)
```



1. Data[0]=1=Data[-8] , Data[1]=2=Data[-7] , Data[2]=3=Data[-6] ,
2. Data[3]=4=Data[-5] , Data[4]=5=Data[-4] , Data[5]=10=Data[-3],
3. Data[6]=19=Data[-2],Data[7]=17=Data[-1]

Tuple Operations

Various Operations can be performed on Tuple. Operations performed on Tuple are given as:

a) Adding Tuple:

Tuple can be added by using the concatenation operator(+) to join two tuples.

Example

```
data1=(1,2,3,4)
data2=('x','y','z')
data3=data1+data2
print data1
print data2
print data3
```

Output:

```
>>>
(1, 2, 3, 4)
('x', 'y', 'z')
(1, 2, 3, 4, 'x', 'y', 'z')
>>>
```

b) Replicating Tuple:

Replicating means repeating. It can be performed by using '*' operator by a specific number of time.

Example

```
tuple1=(10,20,30);
tuple2=(40,50,60);
print tuple1*2
print tuple2*3
```

Output:

```
>>>
(10, 20, 30, 10, 20, 30)
(40, 50, 60, 40, 50, 60, 40, 50, 60)
>>>
```

c) Tuple slicing:

A subpart of a tuple can be retrieved on the basis of index. This subpart is known as tuple slice.

Example

```
data1=(1,2,4,5,7)
print data1[0:2]
print data1[4]
print data1[:-1]
print data1[-5:]
print data1
```

Output:

```
>>>
(1, 2)
7
(1, 2, 4, 5)
(1, 2, 4, 5, 7)
(1, 2, 4, 5, 7)
>>>
```

4.2.1 Tuple assignment

- It is often useful to swap the values of two variables.
- With conventional assignments, you have to use a temporary variable.
- For example, to swap a and b:

```
>>> temp = a
>>> a = b
>>> b = temp
```

- This solution is cumbersome; **tuple assignment** is more elegant:

```
>>> a, b = b, a
```

- The left side is a tuple of variables; the right side is a tuple of expressions.
- Each value is assigned to its respective variable.
- All the expressions on the right side are evaluated before any of the assignments.
- The number of variables on the left and the number of values on the right have to be the same:

```
>>> a, b = 1, 2, 3

ValueError: too many values to unpack
```

- More generally, the right side can be any kind of sequence (string, list or tuple).
- For example, to split an email address into a user name and a domain, you could write:

```
>>> addr = 'monty@python.org'
>>> uname, domain = addr.split('@')
```

- The return value from split is a list with two elements; the first element is assigned to uname, the second to domain.

```
>>> uname
'monty'
>>> domain
'python.org'
```

4.2.2 Tuple as return value

- Strictly speaking, a function can only return one value, but if the value is a tuple, the effect is the same as returning multiple values.
- For example, if you want to divide two integers and compute the quotient and remainder, it is inefficient to compute x/y and then x%y.
- It is better to compute them both at the same time.
- The built-in function divmod takes two arguments and returns a tuple of two values, the quotient and remainder.
- You can store the result as a tuple:

```
>>> t = divmod(7, 3)
>>> t
(2, 1)
```

- Or use tuple assignment to store the elements separately:

```
>>> quot, rem = divmod(7, 3)
>>> quot
2
>>> rem
1
```

- Here is an example of a function that returns a tuple:

```
def min_max(t):
return min(t), max(t)
```

- max and min are built-in functions that find the largest and smallest elements of a sequence.
- min_max computes both and returns a tuple of two values.

Variable-length argument tuples

- Functions can take a variable number of arguments.
- A parameter name that begins with * **gathers** arguments into a tuple.
- For example, print all takes any number of arguments and prints them:

```
def printall(*args):
print(args)
```

- The gather parameter can have any name you like, but args is conventional. Here's how the function works:

```
>>> printall(1, 2.0, '3')
(1, 2.0, '3')
```

- The complement of gather is **scatter**.
- If you have a sequence of values and you want to pass it to a function as multiple arguments, you can use the * operator.
- For example, divmod takes exactly two arguments; it doesn't work with a tuple:

```
>>> t = (7, 3)
>>> divmod(t)
```

```
TypeError: divmod expected 2 arguments, got 1
```

- But if you scatter the tuple, it works:

```
>>> divmod(*t)
(2, 1)
```

- Many of the built-in functions use variable-length argument tuples.
- For example, max and min can take any number of arguments:

```
>>> max(1, 2, 3)
3

But sum does not.
>>> sum(1, 2, 3)

TypeError: sum expected at most 2 arguments, got 3
```

Other Operations:

a) Updating elements in a List:

Elements of the Tuple cannot be updated. This is due to the fact that Tuples are immutable. Whereas the Tuple can be used to form a new Tuple.

Example

```
data=(10,20,30)
data[0]=100
print data
```

Output:

```
>>>
Traceback (most recent call last):
         File "C:/Python27/t.py", line 2, in
        data[0]=100
TypeError: 'tuple' object does not support item assignment
>>>
```

Creating a new Tuple from existing:

Example

```
data1=(10,20,30)
data2=(40,50,60)
data3=data1+data2
print data3
```

Output:

```
>>>
(10, 20, 30, 40, 50, 60)
>>>
```

b) Deleting elements from Tuple:

Deleting individual element from a tuple is not supported. However the whole of the tuple can be deleted using the del statement.

Example

```
data=(10,20,'rahul',40.6,'z')
print data
del data      #will delete the tuple data
print data  #will show an error since tuple data is already deleted
```

Output:

```
>>>
(10, 20, 'rahul', 40.6, 'z')
Traceback (most recent call last):
File "C:/Python27/t.py", line 4, in
print data
NameError: name 'data' is not defined
>>>
```

Functions of Tuple:

There are following in-built Type Functions:

Function	Description
min(tuple)	Returns the minimum value from a tuple.
max(tuple)	Returns the maximum value from the tuple.
len(tuple)	Gives the length of a tuple
cmp(tuple1,tuple2)	Compares the two Tuples.
tuple(sequence)	Converts the sequence into tuple.

1) min(tuple):

Example

```
data=(10,20,'rahul',40.6,'z')
print min(data)
```

Output:

```
>>>
10
>>>
```

2) max(tuple):

Example

```
data=(10,20,'rahul',40.6,'z')
print max(data)
```

Output:

```
>>>
z
>>
```

3) len(tuple):

Example

```
data=(10,20,'rahul',40.6,'z')
print len(data)
```

Output:

```
>>>
5
>>>
```

4) cmp(tuple1,tuple2):

Explanation: If elements are of the same type, perform the comparison and return the result. If elements are different types, check whether they are numbers.

- If numbers, perform comparison.
- If either element is a number, then the other element is returned.
- Otherwise, types are sorted alphabetically .

If we reached the end of one of the lists, the longer list is "larger." If both list are same it returns 0.

Example

```
data1=(10,20,'rahul',40.6,'z')
data2=(20,30,'sachin',50.2)
print cmp(data1,data2)
print cmp(data2,data1)
data3=(20,30,'sachin',50.2)
print cmp(data2,data3)
```

Output:

```
>>>
-1
1
0
>>>
```

5) tuple(sequence):

Example

```
dat=[10,20,30,40]
data=tuple(dat)
print data
```

Output:

```
>>>
(10, 20, 30, 40)
>>>
```

4.3 Dictionaries

- Dictionaries are one of Python's best features;
- A dictionary contains a collection of indices, which are called **keys**, and a collection of values.
- Each key is associated with a single value.
- The association of a key and a value is called a **key-value pair** or sometimes an **item**.
- Dictionary is an unordered set of key and value pair.
- It is an container that contains data, enclosed within curly braces.
- The pair i.e., key and value is known as item.
- The key passed in the item must be unique.
- The key and the value is separated by a colon(:). This pair is known as item.
- Items are separated from each other by a comma(,).
- Different items are enclosed within a curly brace and this forms Dictionary.

Example:

```
data={100:'Ravi' ,101:'Vijay' ,102:'Rahul'}
print data

Output:


>>>
{100: 'Ravi', 101: 'Vijay', 102: 'Rahul'}
>>>
```

- A dictionary is a mapping
- In mathematical language, a dictionary represents a **mapping** from keys to values, so you can also say that each key "maps to" a value.
- The function dict creates a new dictionary with no items.

- Because dict is the name of a built-in function, you should avoid using it as a variable name.

Example

```
>>> eng2sp = dict()

>>> eng2sp
{}
```

- The squiggly-brackets, {}, represent an empty dictionary.
- To add items to the dictionary, you can use square brackets:

```
>>> eng2sp['one'] = 'uno'
```

- This line creates an item that maps from the key 'one' to the value 'uno'.
- If we print the dictionary again, we see a key-value pair with a colon between the key and value:

```
>>> eng2sp
{'one': 'uno'}
```

- This output format is also an input format. For example, you can create a new dictionary with three items:

```
>>> eng2sp = {'one': 'uno', 'two': 'dos', 'three': 'tres'}
```

- But if you print eng2sp, you might be surprised:

```
>>> eng2sp
{'one': 'uno', 'three': 'tres', 'two': 'dos'}
```

- The order of the key-value pairs might not be the same.
- If you type the same example on your computer, you might get a different result.
- In general, the order of items in a dictionary is unpredictable.
- But that's not a problem because the elements of a dictionary are never indexed with integer indices.
- Instead, you use the keys to look up the corresponding values:

```
>>> eng2sp['two']
'dos'
```

- The key 'two' always maps to the value 'dos' so the order of the items doesn't matter.
- If the key isn't in the dictionary, you get an exception:

```
>>> eng2sp['four']
```

```
KeyError: 'four'
```

Mutable

- Dictionary is mutable i.e., value can be updated.
- Key must be unique and immutable.
- Value is accessed by key.
- Value can be updated while key cannot be changed.
- Dictionary is known as Associative array since the Key works as Index and they are decided by the user.

Example:

```
plant={}
plant[1]='Ravi'
plant[2]='Manoj'
plant['name']='Hari'
plant[4]='Om'
print plant[2]
print plant['name']
print plant[1]
print plant
```

Output:

```
>>>
Manoj
Hari
Ravi
{1: 'Ravi', 2: 'Manoj', 4: 'Om', 'name': 'Hari'}
>>>
```

Functions and Methods

Python Dictionary supports the following Functions:

Dictionary Functions:

Functions	Description
len(dictionary)	Gives number of items in a dictionary.
cmp(dictionary1,dictionary2)	Compares the two dictionaries.
str(dictionary)	Gives the string representation of a string.

Dictionary Methods:

Methods	Description
keys()	Return all the keys element of a dictionary.
values()	Return all the values element of a dictionary.
items()	Return all the items(key-value pair) of a dictionary.
update(dictionary2)	It is used to add items of dictionary2 to first dictionary.
clear()	It is used to remove all items of a dictionary. It returns an empty dictionary.
fromkeys(sequence,value1)/ fromkeys(sequence)	It is used to create a new dictionary from the sequence where sequence elements forms the key and all keys share the values ?value1?. In case value1 is not give, it

	set the values of keys to be none.
copy()	It returns an ordered copy of the data.
has_key(key)	It returns a boolean value. True in case if key is present in the dictionary ,else false.
get(key)	Returns the value of the given key. If key is not present it returns none.

Functions:

1) len(dictionary):

Example

```
data={100:'Ram', 101:'Suraj', 102:'Alok'}
print data
print len(data)
```

Output:

```
>>>
{100: 'Ram', 101: 'Suraj', 102: 'Alok'}
3
>>>
```

2) cmp(dictionary1,dictionary2):

Explanation:

The comparison is done on the basis of key and value.

```
If, dictionary1 == dictionary2, returns 0.
      dictionary1 < dictionary2, returns -1.
     dictionary1 > dictionary2, returns 1.
```

Example

```
data1={100:'Ram', 101:'Suraj', 102:'Alok'}
data2={103:'abc', 104:'xyz', 105:'mno'}
data3={'Id':10, 'First':'Aman','Second':'Sharma'}
```

```
data4={100:'Ram', 101:'Suraj', 102:'Alok'}
print cmp(data1,data2)
print cmp(data1,data4)
print cmp(data3,data2)
```

Output:

```
>>>
-1
0
1
>>>
```

3) str(dictionary):

Example

```
data1={100:'Ram', 101:'Suraj', 102:'Alok'}
print str(data1)
```

Output:

```
>>>
{100: 'Ram', 101: 'Suraj', 102: 'Alok'}
>>>
```

Methods:

1) keys():

Example

```
data1={100:'Ram', 101:'Suraj', 102:'Alok'}
print data1.keys()
```

Output:

```
>>>
[100, 101, 102]
>>>
```

2) values():

Example

```
data1={100:'Ram', 101:'Suraj', 102:'Alok'}
print data1.values()
```

Output:

```
>>>
['Ram', 'Suraj', 'Alok']
>>>
```

3) items():

Example

```
data1={100:'Ram', 101:'Suraj', 102:'Alok'}
print data1.items()
```

Output:

```
>>>
[(100, 'Ram'), (101, 'Suraj'), (102, 'Alok')]
>>>
```

4) update(dictionary2):

Example

```
data1={100:'Ram', 101:'Suraj', 102:'Alok'}
data2={103:'Sanjay'}
data1.update(data2)
print data1
print data2
```

Output:

```
>>>
{100: 'Ram', 101: 'Suraj', 102: 'Alok', 103: 'Sanjay'}
{103: 'Sanjay'}
>>>
```

5) clear():

Example

```
data1={100:'Ram', 101:'Suraj', 102:'Alok'}
print data1
data1.clear()
print data1
```

Output:

```
>>>
```

```
{100: 'Ram', 101: 'Suraj', 102: 'Alok'}
{}
>>>
```

6) fromkeys(sequence)/ fromkeys(seq,value):

Example

```
sequence=('Id' , 'Number' , 'Email')
data={}
data1={}
data=data.fromkeys(sequence)
print data
data1=data1.fromkeys(sequence,100)
print data1
```

Output:

```
>>>
{'Email': None, 'Id': None, 'Number': None}
{'Email': 100, 'Id': 100, 'Number': 100}
>>>
```

7) copy():

Example

```
data={'Id':100 , 'Name':'Aakash' , 'Age':23}
data1=data.copy()
print data1
```

Output:

```
>>>
{'Age': 23, 'Id': 100, 'Name': 'Aakash'}
>>>
```

8) has_key(key):

Example

```
data={'Id':100 , 'Name':'Aakash' , 'Age':23}
print data.has_key('Age')
print data.has_key('Email')
```

Output:

```
>>>
True
```

```
False
>>>
```

9) get(key):

Example

```
data={'Id':100 , 'Name':'Aakash' , 'Age':23}
print data.get('Age')
print data.get('Email')
```

Output:

```
>>>
23
None
>>>
```

4.3.1 Operations and methods

The various operations that can be done in dictionary are

Accessing Values

- Since Index is not defined, a Dictionaries value can be accessed by their keys.

Syntax:

```
[key]
```

Example

```
data1={'Id':100, 'Name':'Suresh', 'Profession':'Developer'}
data2={'Id':101, 'Name':'Ramesh', 'Profession':'Trainer'}
print "Id of 1st employer is",data1['Id']
print "Id of 2nd employer is",data2['Id']
print "Name of 1st employer:",data1['Name']
print "Profession of 2nd employer:",data2['Profession']
```

Output:

```
>>>
Id of 1st employer is 100
Id of 2nd employer is 101
Name of 1st employer is Suresh
Profession of 2nd employer is Trainer
>>>
```

Updation

The item i.e., key-value pair can be updated. Updating means new item can be added. The values can be modified.

Example

```
data1={'Id':100, 'Name':'Suresh', 'Profession':'Developer'}
data2={'Id':101, 'Name':'Ramesh', 'Profession':'Trainer'}
data1['Profession']='Manager'
data2['Salary']=20000
data1['Salary']=15000
print data1
print data2
```

Output:

```
>>>
{'Salary': 15000, 'Profession': 'Manager','Id': 100, 'Name': 'Suresh'}
{'Salary': 20000, 'Profession': 'Trainer', 'Id': 101, 'Name': 'Ramesh'}
>>>
```

Deletion

del statement is used for performing deletion operation.

An item can be deleted from a dictionary using the key.

Syntax:

```
del  [key]
```

Whole of the dictionary can also be deleted using the del statement.

Example

```
data={100:'Ram', 101:'Suraj', 102:'Alok'}
del data[102]
print data
del data
print data   #will show an error since dictionary is deleted.
```

Output:

```
>>>
{100: 'Ram', 101: 'Suraj'}
```

```
Traceback (most recent call last):
         File "C:/Python27/dict.py", line 5, in
          print data
NameError: name 'data' is not defined
>>>
```

Looping and dictionaries

- If you use a dictionary in a for statement, it traverses the keys of the dictionary.
- For example, print_hist prints each key and the corresponding value:

```
def print_hist(h):
for c in h:
print(c, h[c])

Here's what the output looks like:

>>> h = histogram('parrot')
>>> print_hist(h)
a 1
p 1
r 2
t 1
o 1
```

- Again, the keys are in no particular order. To traverse the keys in sorted order, you can use the built-in function sorted:

```
>>> for key in sorted(h):
... print(key, h[key])
a 1
o 1
p 1
r 2
t 1
```

Example:

```
Cricket_players={1: "dhoni", 2:"virat",3:"ashwin"}
Cricket_players[4]="Yuvraj"
For i in Cricket_players:
   print("key=",i, \t "value=", Cricket_players[i]
```

```
Output
Key=1     value=dhoni
Key=2    value=virat
Key=3     value=ashwin
Key=4     value=Yuvarj
```

In operator in dictionary

- The in operator works on dictionaries, too; it tells you whether something appears as a key in the dictionary (appearing as a value is not good enough).

```
>>> 'one' in eng2sp
True
>>> 'uno' in eng2sp
False
```

- To see whether something appears as a value in a dictionary, you can use the method values, which returns a collection of values, and then use the in operator:

```
>>> vals = eng2sp.values()
>>> 'uno' in vals
True
```

Reverse lookup

- Given a dictionary d and a key k, it is easy to find the corresponding value v = d[k]. This operation is called a **lookup**.
- But what if you have v and you want to find k? You have face two problems

  1.There might be more than one key that maps to the value v. Depending on the application, you might be able to pick one, or you might have to make a list that contains all of them.

  2. There is no simple syntax to do a **reverse lookup**;

- Here is a function that takes a value and returns the first key that maps to that value:

```
def reverse_lookup(d, v):
for k in d:
   if d[k] == v:
   return k
raise LookupError()
```

- The **raise statement** causes an exception; in this case it causes a LookupError, which is a built-in exception used to indicate that a lookup operation failed.

- If we get to the end of the loop, that means v doesn't appear in the dictionary as a value, so we raise an exception.

Example:

```
mydict = {"Apple": "red", "Banana": "yellow", "Carrot": "orange"}

inverted_dict = dict([[v,k] for k,v in mydict.items()])

print inverted_dict["red"]
```

- Here is an example of a successful reverse lookup:

```
>>> h = histogram('parrot')
>>> key = reverse_lookup(h, 2)
>>> key
'r'
```

- And an unsuccessful one:

```
>>> key = reverse_lookup(h, 3)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
File "<stdin>", line 5, in reverse_lookup

LookupError
```

4.4 Advanced list processing-

list comprehension

- Python includes a more advanced and powerful operation known as a list comprehension expression.
- List comprehensions are coded in square brackets and are composed of an expression and a looping construct that share a variable name
- The output of list comprehension is List

Normal way of coding a operation using for loop

Example.1

```
>>> res = []
>>> for c in 'SPAM':
...         res.append(c * 4)
 ...
>>> res
 ['SSSS', 'PPPP', 'AAAA', 'MMMM']
```

- Obtaining the same output using List comprehension

```
>>> res = [c * 4 for c in 'SPAM']
>>>res
['SSSS', 'PPPP', 'AAAA', 'MMMM']
```

- From the above examples we can find the power of List comprehension.
- In the normal we used three lines of code to obtain a result.
- But by using List Comprehension we used only one line to obtain the same result.
- List comprehensions are sequence operations, they always build new lists, but they may be used to iterate over any sequence objects, including tuples, strings, files,and other lists.

Example 2

```
>>> T = (1, 2, 3, 4, 5)
 >>> L = [x + 20 for x in T]
>>> L
[21, 22, 23, 24, 25]
```

- List comprehension consists of square brackets containing an expression followed by a for clause, then we can also use more for or if clauses

Example 3

```
>>> seq1 = "spam"
 >>> seq2 = "scam"
>>> a = []
>>> for i in seq1:
...       if x in seq2:
...           res.append(x)
...
>>> res
 ['s', 'a', 'm']
```

- Now we will use List Comprehension for the same output

```
>>> [x for x in seq1 if x in seq2]
['s', 'a', 'm']
```

Example 4

- Printing prime numbers using List comprehension

```
>>> noprimes = [j for i in range(2, 8) for j in range(i*2, 50, i)]
>>> primes = [x for x in range(2, 50) if x not in noprimes]
>>> print primes
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]
```

4.5 Illustrative programs

4.5.1 Selection sort

```
def selectionSort(x):
   for i in range(len(x)-1,0,-1):
      pMax=0
      for j in range(1,i+1):
         if x[j]>x[pMax]:
            pMax = j

      tmp = x[i]
      x[i] = x[pMax]
      x[pMax] = tmp

x = [98,26,52,21,67,39,48,99,11]
selectionSort(x)
print(x)
```

4.5.2 Insertion sort

```
def insertionSort(x):
   for index in range(1,len(x)):
     currentvalue = x[index]
     position = index
     while position>0 and x[position-1]>currentvalue:
```

```
        x[position]=x[position-1]
        position = position-1
    x[position]=currentvalue
x = [98,26,52,21,67,39,48,99,11]
insertionSort(x)
print(x)
```

4.5.3 Merge sort

```
def mergeSort(x):
    print("Splitting ",x)
    if len(x)>1:
        mid = len(x)//2
        lefthalf = x[:mid]
        righthalf = x[mid:]

        mergeSort(lefthalf)
        mergeSort(righthalf)

        i=0
        j=0
        k=0
        while i < len(lefthalf) and j < len(righthalf):
            if lefthalf[i] < righthalf[j]:
                x[k]=lefthalf[i]
                i=i+1
            else:
                x[k]=righthalf[j]
                j=j+1
            k=k+1

        while i < len(lefthalf):
            x[k]=lefthalf[i]
            i=i+1
            k=k+1

        while j < len(righthalf):
            x[k]=righthalf[j]
            j=j+1
            k=k+1
    print("Merging ",alist)

x = [98,26,52,21,67,39,48,99,11]
mergeSort(x)
print(x)
```

4.5.4 Histogram

```
def histogram( items ):
    for n in items:
        output = ''
        times = n
        while( times > 0 ):
          output += '*'
          times = times - 1
        print(output)

histogram([2, 3, 6, 5])
```

Part A

1. Define Lists?

Lists are the most versatile of Python's compound data types. A list contains items separated by commas and enclosed within square brackets ([]). To some extent, lists are similar to arrays in C. One difference between them is that all the items belonging to a list can be of different data type.

2. Define Tuples?

A tuple is another sequence data type that is similar to the list. A tuple is a sequence of immutable Python objects. A tuple consists of a number of values separated by commas. Unlike lists, however, tuples are enclosed within parentheses.

3. Difference between lists and tuples?

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.

4. Define Dictionary?

Python's dictionaries are kind of hash table type. They work like associative arrays or hashes found in Perl and consist of key-value pairs. A dictionary key can be almost any Python type, but are usually numbers or strings. Values, on the other hand, can be any arbitrary Python object.

Dictionaries are enclosed by curly braces ({ }) and values can be assigned and accessed using square braces ([]).

5. What are the properties of Dictionary keys?

1. More than one entry per key not allowed.

2. Keys must be immutable.

6. Mention the built in functions with tuple.

All(), any(), enumerate(), len(), max(), min(), sorted(), tuple(), sum() are the built in functions used with tuple to perform different tasks.

7. What is python dictionary comprehension.

Dictionary Comprehension is an elegant and concise way to create new dictionary from an iterable in python.

Dictionary comprehension consists of an expression pair (key: value) followed by for statement inside curly braces{}.

Dictionary comprehension can optionally contain more for or if statements. An optional if statement can filter out items to form the new dictionary.

8. Mention the built in functions with dictionary.

all(), any(), len(), cmp(), sorted() are the built in functions used with dictionary to perform different tasks.

9. What are the advantages of Tuple over List?

1. We generally use tuple for heterogeneous(different) data types and list for homogeneous (similar) data types.

2. Since tuple are immutable, iterating through tuple is faster than with list. So there is a slight performance boost.

3. Tuples that contain immutable elements can be used as key for a dictionary. With list, this is not possible.

4. If you have data that doesn't change, implementing it as tuple will guarantee that it remains write-protected.

10. What are the basic list operations?

Lists respond to the + and * operators much like strings; they mean concatenation and repetition here too, except that the result is a new list, not a string.

11. What are the methods available in python list?

Append(), extend(), insert(), remove(), pop(), clear(), index(), count(), sort(), reverse(), copy() are the methods available in python list.

12. What is Membership operator? What are its types?

Python's membership operators test for membership in a sequence, such as strings, lists or tuples.

There are two membership operators.

1. in

2. not in

13. What are the three steps of quicksort?

1. Divide

2. Conquer

3. Combine

14. How is tuple created?

A Tuple is created by placing all the items(elements) inside a parantheses(), separated by comma. The parantheses are optional but is a good practice to write it. A tuple can have any number of items and they may be of different types(integer, float, list, string etc..).

15. What are the various ways to access elements in a tuple?

The various ways in which we can access the elements of a tuple.

1. Indexing

2. Negative Indexing

3. Slicing

4. Changing a tuple

Part B

1. Define Lists. Explain in detail.
2. Explain the list operations and the list slices indetail with examples.
3. What are the list methods. Discuss briefly.
4. Explain list loop, mutability, aliasing briefly
5. What is meant by cloning lists.Discuss in detail
6. What are the list parameters.
7. Explain each
8. Define Tuples. Discuss tuple in detail
9. How are the tuple assignment done in python. Justify your answer with examples.
10. What is meant by Dictionaries in python.
11. List the operations and methods of dictionaries and discuss about each in detail.
12. Explain advanced list processing
13. Explain list comprehension;
14. Illustrative programs: selection sort
15. Illustrative programs: insertion sort
16. Illustrative programs: merge sort
17. Illustrative programs: histogram.

Glossary

list: A sequence of values.

element: One of the values in a list (or other sequence), also called items.

nested list: A list that is an element of another list.

accumulator: A variable used in a loop to add up or accumulate a result.

augmented assignment: A statement that updates the value of a variable using an operator like +=.

reduce: A processing pattern that traverses a sequence and accumulates the elements into a single result.

map: A processing pattern that traverses a sequence and performs an operation on each element.

filter: A processing pattern that traverses a list and selects the elements that satisfy some criterion.

object: Something a variable can refer to. An object has a type and a value.

equivalent: Having the same value.

identical: Being the same object (which implies equivalence).

reference: The association between a variable and its value.

aliasing: A circumstance where two or more variables refer to the same object.

delimiter: A character or string used to indicate where a string should be split.

tuple: An immutable sequence of elements.

tuple assignment: An assignment with a sequence on the right side and a tuple of variables on the left. The right side is evaluated and then its elements are assigned to the variables on the left.

gather: The operation of assembling a variable-length argument tuple.

scatter: The operation of treating a sequence as a list of arguments.

zip object: The result of calling a built-in function zip; an object that iterates through a sequence of tuples.

iterator: An object that can iterate through a sequence, but which does not provide list operators and methods.

data structure: A collection of related values, often organized in lists, dictionaries, tuples, etc.

shape error: An error caused because a value has the wrong shape; that is, the wrong type or size.

mapping: A relationship in which each element of one set corresponds to an element of another set.

dictionary: A mapping from keys to their corresponding values.

key-value pair: The representation of the mapping from a key to a value.

item: In a dictionary, another name for a key-value pair.

key: An object that appears in a dictionary as the first part of a key-value pair.

value: An object that appears in a dictionary as the second part of a key-value pair. This is more specific than our previous use of the word "value".

implementation: A way of performing a computation.

hashtable: The algorithm used to implement Python dictionaries.

hash function: A function used by a hashtable to compute the location for a key.

hashable: Atype that has a hash function. Immutable types like integers, floats and strings are hashable; mutable types like lists and dictionaries are not.

lookup: A dictionary operation that takes a key and finds the corresponding value.

reverse lookup: A dictionary operation that takes a value and finds one or more keys that map to it.

raise statement: A statement that (deliberately) raises an exception.

singleton: A list (or other sequence) with a single element.

call graph: A diagram that shows every frame created during the execution of a program, with an arrow from each caller to each callee.

memo: A computed value stored to avoid unnecessary future computation.

global variable: A variable defined outside a function. Global variables can be accessed from any function.

flag: A boolean variable used to indicate whether a condition is true.

declaration: A statement like global that tells the interpreter something about a variable.