UNIT – 4

COMPOUND DATA: LISTS, TUPLES, DICTIONARIES

4.1 LISTS

A list is a sequence of any type of values and can be created as a set of comma-separated values within square brackets. The values in a list are called elements or items. A list within another list is called nested list.

Sample code for creating lists

```
list1 = [‘Ram’, ‘Chennai’, 2017]          # list of different types of elements
list2 = [10, 20, 30, 40, 50]              # list of numbers
list3 = []                                # empty list
list4 = [‘Priya’, 2017, 99.8, [‘Mumbai’, ‘India’]]          # nested list
print list1
print list2, list3
print list4
```

Sample Output:

```
[‘Ram’, ‘Chennai’, 2017]
[10, 20, 30, 40, 50] [ ]
[‘Priya’, 2017, 99.8, [‘Mumbai’, ‘India’]]
```

4.1.1 Accessing elements in lists using subscript operator

The indices of list’s elements are numbered from 0 from the left end and numbered from -1 from the right end. For a list [‘Ram’, ‘Chennai’, 2017], the indices of elements ‘Ram’, ‘Chennai’ and 2017 are shown in the following figure:

Element	‘Ram’	‘Chennai’	2017
Index from left end	0	1	2
Index from right end	–3	–2	–1

To access the element(s) of a list, subscript operator [ ] (also known as slicing operator) is used. Index within [ ] indicates the position of the particular element in the list and it must be an integer expression.

For eg, in a list stulist = [‘Ram’, ‘Chennai’, 2017], stulist[1] returns Chennai as its output.

4.1.2 List operations

Lists operate on + and * operators. Here, + represents concatenation and * represents repetition. The following code explains the concatenation of two lists named num1 and num2:

Sample Code and output for + (Concatenation)

Code	Output

num1=[10, 20, 30]
num2=[40, 50]
num3=num1+num2
print num1
print num2
print num3
	[10, 20, 30]
	[40, 50]
	[10, 20, 30, 40, 50]

The following code is another example for concatenation, where a list contains elements of different data types:

Code	Output

stulist = [‘Ram’, ‘Chennai’, 2017]
newlist = stulist+[‘CSE’]
print stulist
print newlist
	[‘Ram’, ‘Chennai’, 2017]
	[‘Ram’, ‘Chennai’, 2017, ‘CSE’]

Sample code and output for * (Repetition)

The following code describes the repetition operation which is performed on a list num1 for 3 times:

Code	Output

num1=[10, 20]
num2=num1*3
print num1
print num2
	[10, 20]
	[10, 20, 10, 20, 10, 20]

4.1.3 List Slices

A part of a list is called list slice. The operator [m:n] returns the part of the list from mth index to nth index, including the element at mth index but excluding the element at nth index.

If the first index is omitted, the slice starts at the beginning of the string. If the second index is omitted, the slice goes to the end of the string. If the first index is greater than or equals to the second, the slice is an empty string. If both indices are omitted, the slice is a given string itself.

Sample code and output for list slicing

Code	Output	Description
stulist = [‘Ram’, ‘Chennai’, 2017]		List is created with 3 elements
print stulist[0]	Ram	Slice has the element at index 0
print stulist[:3]	[‘Ram’, ‘Chennai’, 2017]	Slice is from the beginning
print stulist[1:]	[‘Chennai’, 2017]	Slice goes to the end
print stulist[1:1]	[]	Slice is empty
print stulist[5:2]	[]	Slice is empty
print stulist[:]	[‘Ram’, ‘Chennai’, 2017]	Entire list is the slice
print stulist[-2:]	[‘Chennai’, 2017]	Slice goes to the end
print stulist[:-2]	[‘Ram’]	Slice is from the beginning
print stulist[1:3]	[‘Chennai’, 2017]	Slice is from 1st index to 2nd index (excluding the 3rd index)

4.1.4 List methods

Python provides the following methods that work on lists:

1. append

Adds element to the end of specified list and does not return any value.

Syntax: listname.append(element)

Sample code for append

```
stulist = [‘Ram’, ‘Chennai’, 2017]
stulist.append(‘CSE’)
print ‘After appending’
print stulist
```

Sample output:

```
After appending
[‘Ram’, ‘Chennai’, 2017, ‘CSE’]
```

2. count

Returns the number of occurrences of an element in a specified list.

Syntax: listname.count(element)

Sample code for count

```
stulist = [‘Ram’, ‘Chennai’, 2017, ‘Priya’, ‘Mumbai’, 2017]
print stulist
print 'Count for Chennai : ', stulist.count('Chennai')
print 'Count for 2017 : ', stulist.count(2017)
```

Sample output:

```
['Ram', 'Chennai', 2017, 'Priya', 'Mumbai', 2017]
Count for Chennai :  1
Count for 2017 :  2
```

3. extend

Appends the contents of secondlist to the firstlist and does not return any value.

Syntax: firstlist.extend(secondlist)

Sample code for extend

```
stulist = ['Ram', 'Chennai', 2017]
dept = ['CSE']
print "Before Extend : ", stulist
stulist.extend(dept)
print "After Extend : ", stulist
```

Sample output:

```
Before Extend :  ['Ram', 'Chennai', 2017]
After Extend :  ['Ram', 'Chennai', 2017, 'CSE']
```

4. index

Returns the index of an element, if an element is found in the specified list. Else, an exception is raised.

Syntax: listname.index(element)

Sample code for index

```
stulist = ['Ram', 'Chennai', 2017]
print 'Index of Ram : ', stulist.index( 'Ram' )
print 'Index of Chennai : ', stulist.index( 'Chennai' )
print 'Index of 2017 : ', stulist.index(2017)
```

Sample output:

```
Index of Ram :  0
Index of Chennai :  1
Index of 2017 :  2
```

6. insert

Inserts the given element at the given index in a specified list and does not return any value

Syntax: listname.insert(index, element)

Sample code for insert

```
stulist = [‘Ram’, ‘Chennai’, 2017]
print ‘Before insert : ‘,stulist
stulist.insert(1, ‘CSE’)
print ‘After insert : ‘, stulist
```

Sample output:

```
Before insert :  [‘Ram’, ‘Chennai’, 2017]
After insert :  [‘Ram’, ‘CSE’, ‘Chennai’, 2017]
```

7. pop

Removes and returns the element from the end of specified list

Syntax: listname.pop()

Sample code for pop

```
stulist = [‘Ram’, ‘Chennai’, 2017, ‘CSE’, 92.7]
print ‘Initial list is : ‘, stulist
print ‘Popping the last item : ‘, stulist.pop()
print ‘After popping the last item, the list is : ‘, stulist
```

Sample output:

```
Initial list is :  [‘Ram’, ‘Chennai’, 2017, ‘CSE’, 92.7]
Popping the last item :  92.7
After popping the last item, the list is :  [‘Ram’, ‘Chennai’, 2017, ‘CSE’]
```

8. pop(index)

Removes and returns the element at given index.

Syntax: listname.pop(index)

Sample code for pop(index)

```
stulist = [‘Ram’, ‘Chennai’, 2017, ‘CSE’, 92.7]
print ‘Initial list is : ‘, stulist
print ‘Popping an item with index 2 : ‘, stulist.pop(2)
```

```
# 2 is an index of the item to be removed
print 'Now the list is : ',stulist
```

Sample output:

```
Initial list is :  ['Ram', 'Chennai', 2017, 'CSE', 92.7]
Popping an item with index 2 :  2017
Now the list is :  ['Ram', 'Chennai', 'CSE', 92.7]
```

9. *remove*

Removes an element from the list and does not return any value.

Syntax: listname.remove(element)

Sample code for remove

```
stulist = ['Ram', 'Chennai', 2017, 'CSE', 92.7, 2017]
print 'Initial list is : ', stulist
stulist.remove('CSE')
print 'After removing CSE from the list : ', stulist
stulist.remove(2017)
print 'After removing 2017 from the list : ', stulist
```

Sample output:

```
Initial list is :  ['Ram', 'Chennai', 2017, 'CSE', 92.7, 2017]
After removing CSE from the list :  ['Ram', 'Chennai', 2017, 92.7, 2017]
After removing 2017 from the list :  ['Ram', 'Chennai', 92.7, 2017]
```

10. *reverse*

Reverses the entire list

Syntax: listname.reverse()

Sample code for reverse

```
stulist = ['Ram', 'Chennai', 2017, 'CSE', 92.7]
print 'Initial list is : ', stulist
stulist.reverse()
print 'After reversing, the list is : ', stulist
```

Sample output:

```
Initial list is :  ['Ram', 'Chennai', 2017, 'CSE', 92.7]
After reversing, the list is :  [92.7, 'CSE', 2017, 'Chennai', 'Ram']
```

11. sort

Sorts the list in ascending order.

Syntax: listname.sort()

Sample code for sort

```
numlist = [6, 28, 11, 4, 20, 26, 13, 12]
print 'Before sorting : ', numlist
numlist.sort()
print 'After sorting is : ', numlist
```

Sample output:

```
Before sorting :  [6, 28, 11, 4, 20, 26, 13, 12]
After sorting is :  [4, 6, 11, 12, 13, 20, 26, 28]
stulist = ['Ram', 'Chennai', 2017, 'CSE', 92.7]
print 'Initial list is : ', stulist
stulist.sort()
print 'After sorting, the list is : ', stulist
```

Sample output:

```
Initial list is :  ['Ram', 'Chennai', 2017, 'CSE', 92.7]
After sorting, the list is :  [92.7, 2017, 'CSE', 'Chennai', 'Ram']
```

4.1.5 List Loop

The general way to traverse the elements of a list is with *for* loop. The following code shows the use of for loop in accessing the elements of a list.

```
numlist = [1, 2, 3, 4, 5]
    for i in numlist:
        print i
```

Output:

```
1
2
3
4
5
```

The following code illustrates the use of range and *len* functions in accessing the indices of the elements of a list:

```
numlist = [1, 2, 3, 4, 5]
```

```
for i in range(len(numlist)):
    print i
```

Output:

```
0
1
2
3
4
```

Here, *len* is a function that returns the number of elements in the list and range is a function that returns a list of indices from 0 to $n - 1$, where n is the length of the list. The following code gives an idea to traverse the list and to update the elements of a list with the help of range and *len* functions in for loop:

```
numlist = [1, 2, 3, 4, 5]
for i in range(len(numlist)):
    numlist[i]=numlist[i]+10
for i in numlist:
    print i
```

Output:

```
11
12
13
14
15
```

A for loop over an empty list never executes the body and is shown in the following code:

```
numlist = []
for i in numlist:
    print 'never executes'
```

4.1.6 Mutability

The list is a **mutable data structure**. This means that its elements can be replaced, inserted and removed. A slice operator on the left side of an assignment operation can update single or multiple elements of a list. New elements can be added to the list using append() method.

The following code replaces 'Ram' which is at index 0 in the stulist by 'Priya'. The values are shown in the output for both instances.

```
stulist = ['Ram', 'Chennai', 2017]
print 'Before mutation ', stulist
```

```
stulist[0] = ‘Priya’
print ‘After mutation ‘, stulist
```

Output:

```
        Before mutation  [‘Ram’, ‘Chennai’, 2017]
        After mutation  [‘Priya’, ‘Chennai’, 2017]
```

4.1.7 Aliasing

When we create two lists, we get two objects as shown in the following code and the corresponding state diagram (Figure 4.1):

```
list1=[1,2]
list2=[1,2]
print list1 is list2        # prints False, as list1 and list2 are not the same object
```

$$\begin{tikzcd}
\text{List 1} \arrow[r] & {[1,2]} \\
\text{List 2} \arrow[r] & {[1,2]}
\end{tikzcd}$$

Figure 4.1. State Diagram

Here, the two lists *list1* and *list2* are equivalent since they have the same elements. But they are not identical since they are not the same object. If two objects are identical, then they are equivalent. But if two objects are equivalent, then they are not necessarily identical.

In the following code, 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.

```
list1=[1,2]
list2=list1
print list1 is list2        # prints True, as list1 and list2 are the same object
```

The state diagram for the above code is as shown in Figure 4.2:

$$\begin{tikzcd}
\text{List 1} \arrow[dr] & \\
 & {[1,2]} \\
\text{List 2} \arrow[ur] &
\end{tikzcd}$$

Figure 4.2. State Diagram

If the aliased object is mutable, modifications done in one object affect the other object also. In the following code, list1 and list2 are aliased objects. Changes made in list1 affect list2 and similarly, changes done in list2 affect list1.

Sample Code

```
list1=[1,2]
list2=list1
print 'List1 is :', list1
print 'List2 is :', list2
list1[0]=10
print 'List1 is :', list1
print 'List2 is :', list2
list2[1]=20
print 'List1 is :', list1
print 'List2 is :', list2
```

Sample Output:

```
List1 is : [1, 2]
List2 is : [1, 2]
List1 is : [10, 2]
List2 is : [10, 2]
List1 is : [10, 20]
List2 is : [10, 20]
```

Though aliasing can be helpful, it may lead to errors. So, avoid aliasing in mutable objects. To prevent aliasing in lists, a new empty list can be created and the contents of the existing list can be copied to it, as given in the following code:

```
list1=[1,2]                    # Existing list
list2=[]                       # New and Empty list
for e in list1:
   list2.append(e)
print 'List1 is :', list1
print 'List2 is :', list2
list1[0]=10
print 'After modification'
print 'List1 is :', list1
print 'List2 is :', list2
```

Output:

```
List1 is : [1, 2]
List2 is : [1, 2]
After modification
List1 is : [10, 2]
List2 is : [1, 2]
```

4.1.8 Cloning Lists

Assignment statements in Python do not copy objects. They simply create bindings between two objects. For mutable sequences (like lists), a copy of an existing object may be required so that one object can be changed without affecting another.

In lists, cloning operation can be used to create a copy of an existing list so that changes made in one copy of list will not affect another. The copy contains the same elements as the original.

Method 1: `list()` ***function:***

Built-in list() function can be used for cloning lists with the following syntax:

Newlistname = list(Oldlistname)

Sample Code:

```
oldlist = [10, 20, 30, 40, 50]
newlist = list(oldlist)
print ‘Old list is : ‘, oldlist
print ‘New list is : ‘, newlist
oldlist[0]=5
print ‘Old list is : ‘, oldlist
print ‘New list is : ‘, newlist
```

Sample Output:

```
Old list is :  [10, 20, 30, 40, 50]
New list is :  [10, 20, 30, 40, 50]
Old list is :  [5, 20, 30, 40, 50]
New list is :  [10, 20, 30, 40, 50]
```

Method 2: copy.copy() function:

Syntax:

Newlistname = copy.copy(Oldlistname)

copy.copy() is little slower than `list()` since it has to determine data type of `old list` first.

Sample Code:

```
import copy
oldlist = [10, 20, 30, 40, 50]
newlist = copy.copy(oldlist)        # Returns a shallow copy of oldlist
print 'Old list is : ', oldlist
print 'New list is : ', newlist
oldlist[0]=5
print 'Old list is : ', oldlist
print 'New list is : ', newlist
```

Sample Output:

```
Old list is :  [10, 20, 30, 40, 50]
New list is :  [10, 20, 30, 40, 50]
Old list is :  [5, 20, 30, 40, 50]
New list is :  [10, 20, 30, 40, 50]
```

Method 3: copy.deepcopy() function:

Syntax

```
Newlistname = copy.deepcopy(Oldlistname)
```

copy.deepcopy() is the slowest and memory-consuming method.

Sample Code:

```
import copy
oldlist = [10, 20, 30, 40, 50]
newlist = copy.deepcopy(oldlist)            # Returns a deep copy of oldlist
print 'Old list is : ', oldlist
print 'New list is : ', newlist
oldlist[0]=5
print 'Old list is : ', oldlist
print 'New list is : ', newlist
```

Sample Output:

```
Old list is :  [10, 20, 30, 40, 50]
New list is :  [10, 20, 30, 40, 50]
Old list is :  [5, 20, 30, 40, 50]
New list is :  [10, 20, 30, 40, 50]
```

copy() (also known as shallow copy) and *deepcopy()* differs in the usage of compound objects that are objects containing other objects, like lists). *copy()* creates a new compound object first and then inserts references to the objects of the original. *deepcopy()* constructs a new compound object and then, recursively, inserts copies to the objects of the original. The following code illustrates the use of *deepcopy()* for a compound (nested) list.

Sample Code:

```
import copy
oldlist = [1, 2, [‘a’,’b’]]
newlist = copy.deepcopy(oldlist)
print ‘Old list is : ‘, oldlist
print ‘New list is : ‘, newlist
newlist[0] = ‘c’
newlist[2][1] = ‘d’
print ‘Old list is : ‘, oldlist
print ‘New list is : ‘, newlist
```

Sample Output:

```
Old list is :  [1, 2, [‘a’, ‘b’]]
New list is :  [1, 2, [‘a’, ‘b’]]
Old list is :  [1, 2, [‘a’, ‘b’]]
New list is :  [‘c’, 2, [‘a’, ‘d’]]
```

4.1.9 List parameters

When a list is passed as a parameter to a function, the function gets a reference to the list. In the following code, *numlist* is a list and it is passed as a parameter to my_insert() function. Within my_insert(), it is referenced as *t*.

```
def my_insert(t):     # function definition
   t.insert(1,15)
numlist = [10, 20, 30, 40, 50]
print ‘Before calling my_insert function : ‘, numlist
my_insert(numlist)      # function call
print ‘After calling my_insert function : ‘, numlist
```

Here, the parameter *t* and the variable *numlist* are aliases for the same object. my_insert() function inserts a new element 15 at index 1 in the list. This change is visible to the caller. The elements of a list before and after calling my_insert() are given below as the output:

```
Before calling my_insert function :  [10, 20, 30, 40, 50]
After calling my_insert function :  [10, 15, 20, 30, 40, 50]
```

The following program employs a function *my_display()* that creates and returns a new list. Within *my_display()*, *numlist* is referenced as n.

Sample Code:

```
def my_display(n):     # function definition
   return n[:]
   numlist = [10, 20, 30, 40, 50]
print ‘numlist is : ‘, numlist
newlist=my_display(numlist)      # function call
print ‘newlist is : ‘, newlist
```

Sample Output:

```
numlist is :  [10, 20, 30, 40, 50]
newlist is :  [10, 20, 30, 40, 50]
```

The following program includes a function *my_display()* that creates and displays the elements of a list.

Sample Code:

```
def my_display(n):     # function definition
   nlist= n[:]
   print ‘Within a function : ‘, nlist
   numlist = [10, 20, 30, 40, 50]
print ‘numlist is : ‘, numlist
my_display(numlist)      # function call
```

Sample Output:

```
numlist is :  [10, 20, 30, 40, 50]
Within a function :  [10, 20, 30, 40, 50]
```

4.1.10 Deleting list elements

To remove a list element, *del* operator can be used if an element to be deleted is known. In the following code, the element ‘Chennai’ is deleted by mentioning its index in the del operator.

```
stulist = [‘Ram’, ‘Chennai’, 2017, ‘CSE’, 92.7]
print ‘Initial list is : ‘, stulist
```

```
del stulist[1]
print 'Now the list is : ', stulist
```

Output:

```
Initial list is :  ['Ram', 'Chennai', 2017, 'CSE', 92.7]
Now the list is :  ['Ram', 2017, 'CSE', 92.7]
```

pop() and *remove()* methods can also be used to delete list elements.

4.1.11 Python functions for list operations

1. cmp

Compares two lists and returns 0, if they are equal. Else, returns a nonzero vale.

Syntax: cmp(list1, list2)

Code	Output

list1 = [10, 'xyz']
list2 = [20, 'abc']
list3 = [10, 'xyz']	

print 'Comparing list1 and list2 ', cmp(list1, list2) print 'Comparing list2 and list3 ', cmp(list2, list3) print 'Comparing list1 and list3 ',cmp(list1, list3)	Comparing list1 and list2 -1 Comparing list2 and list3 1 Comparing list1 and list3 0

1. ***len***

Returns the total number of elements in a list.

Syntax: len(listname)

Code	Output
stulist = ['Ram', 'Chennai', 2017, 'CSE', 92.7]	
print 'Length is : ', len(stulist)	Length is : 5

2. max

Returns the largest item from the list

Syntax: max(listname)

3. min

Returns the smallest item from the list

Syntax: min(listname)

Code	Output
numlist = [6, 28, 11, 4, 20, 26, 13, 12] print ‘Maximum is : ‘, max(numlist) print ‘Minimum is : ‘, min(numlist)	 Maximum is : 28 Minimum is : 4
stulist = [‘Anu’, ‘Chennai’, 2017, ‘CSE’, 92.7] print ‘Maximum is : ‘, max(stulist) print ‘Minimum is : ‘, min(stulist)	 Maximum is : Chennai Minimum is : 92.7

4. list

Converts a tuple into a list and returns a list.

Syntax: listname=list(tuplename)

Code	Output
stu_tuple = (‘Anu’, ‘Chennai’, 2017, ‘CSE’, 92.7) print ‘Tuple elements : ‘, stu_tuple	Tuple elements : (‘Anu’, ‘Chennai’, 2017, ‘CSE’, 92.7)
stulist = list(stu_tuple) print ‘List elements : ‘, stulist	List elements : [‘Anu’, ‘Chennai’, 2017, ‘CSE’, 92.7]

4.1.12 List Comprehension

Comprehensions are constructs that allow sequences to be built from other sequences. It provides a concise way to create lists. Python 2.0 introduced list comprehensions and Python 3.0 comes with dictionary and set comprehensions.

A list comprehension consists of the following parts:

- An Input Sequence.
- Variable representing members of the input sequence.
- An Optional Predicate expression.
- An Output Expression producing elements of the output list from members of the Input Sequence that satisfy the predicate.

Syntax:

```
[expression for item in list if conditional]
```

This is equivalent to:

```
for item in list:
        if conditional:
                expression
new_list = [expression(i) for i in old_list if filter(i)]
```

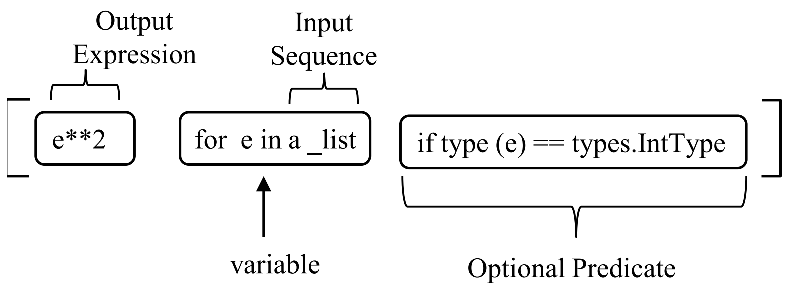
new_list is the resultant list. *expression(i)* is based on the variable used for each element in the old list. If needed *filter* can be applied using if statement.

Example:

```
    from types import *
a_list = [1, '4', 9, 'a', 0, 4]
squared_ints = [ e**2 for e in a_list if type(e) == IntType ]
print squared_ints
```

Sample output:

```
[ 1, 81, 0, 16 ]
```



- The iterator part iterates through each member **e** of the input sequence **a_list**.
- The predicate checks if the member is an integer.
- If the member is an integer then it is passed to the output expression, squared, to become a member of the output list.

Example:

```
x=[I for I in range(10)]
print x
```

Sample Output:

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
squares=[x**2 for x in range(10)]
print squares
```

Sample Output:

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

```
str=["this", "is", "an", "example"]
items=[word[0] for word in str]
print items
```

Sample Output:

```
['t', 'i', 'a', 'e']
```

```
value=[x+y for x in [10,20,30] for y in [1,2,3]]
print value
```

Sample Output:

```
[11, 12, 13, 21, 22, 23, 31, 32, 33]
```

This example, adds the values in list *x* to each value in list *y*.

4.2 TUPLES

A tuple is a collection of values of different types. Unlike lists, tuple values are indexed by integers. The important difference is that tuples are immutable.

4.2.1 Advantages of Tuple over List

Tuples are like lists, so both of them are used in similar situations as well. However, there are specific advantages of implementing a tuple over a list. The advantages of tuples over list are listed below.

- Tuples generally used for heterogeneous (different) datatypes and list for homogeneous (similar) datatypes.
- Tuples are immutable, so iterating through tuple is faster than with list. There is a slight performance enhancement through list.
- Tuple elements can be used as key for a dictionary. With list, this is not possible.
- If you have data that doesn't change, implementing it as tuple will guarantee that it remains write-protected.

As we have seen before, a tuple is a comma-separated values.

Syntactically, a tuple can be represented like this:

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

Even if it is not necessary to parentheses to enclose tuples, it is so.

```
t1 = ('a', 'b', 'c', 'd', 'e')
t2=(1,2,3,4,5)
```

The empty tuple can be created by simply using the parentheses.

```
t3=()
```

The tuple with one element can be created as follows. It takes one comma after the element.

```
t4=('s',)
print  type(t4)
```

Output:

```
<type 'tuple'>
```

```
t5 = ‘a’,
print type(t5)
```

Output:

```
<type ‘tuple’>
```

A value of tuple in parentheses is not a tuple. The following program code explain this.

```
t2 = (‘a’)
print type(t2[1])
```

Output:

```
<type ‘str’>
```

The built –in function tuple can be used to create tuple. To create an empty tuple no arguments is passed in the built-in function.

```
t = tuple()        #tuple is the built-in function
print t
```

Output:

```
()
```

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

```
t = tuple(‘hello’)

print t
```

Output:

```
(‘h’, ‘e’, ‘l’, ‘l’, ‘o’)
```

```
t = tuple(’12345’)
print t
```

Output:

```
(‘1’, ‘2’, ‘3’, ‘4’, ‘5’)
```

Program to illustrate the tuple creation for single element

```
# only parentheses is not enough
tup1 = (“hai”)
print type(tup1)
# need a comma at the end
```

```
tup2 = (“hai”,)
print type(tup2)
# parentheses is optional
tup3 = “hai”,
print type(tup3)
```

Output:

```
<class ‘str’>
<class ‘tuple’>
<class ‘tuple’>
```

4.2.2 Accessing values

To access the tuple elements slicing (bracket operator [ ]) operator along with index or indices is used.

```
t1 = (‘C’, ‘C++’, ‘python’, 1997, 2000);
t2 = (1, 2, 3, 4, 5, 6, 7 );
t3= (‘a’, ‘b’, ‘c’, ‘d’, ‘e’)
print “tup1[0]: “, tup1[0]
print “tup1[1]: “, tup1[1]
print “tup2[1:5]: “, tup2[1:5]
print “tup2[1:]: “, tup2[1:]
print t[0]
```

Output:

```
tup1[0]:  C
tup1[1]: C++
tup2[1:5]:  [2, 3, 4, 5, 6, 7]
a
```

Program to illustrate the accessing the tuple elements

```
t1 = [‘p’,’y’,’t’,’h’,’o’,’n’]
print t1[0]
print t1[5]
print t1[2]
print t1[-1]
print t1[-6]
```

```
# TypeError: list indices must be integers, not float
#t1[2.0]
# nested tuple
nest_tup = ("hello", [8, 4, 6], (1, 2, 3))
# nested index
print nest_tup[0][4]
# nested index
# Output: 4
print nest_tup[1][2]
print nest_tup[2][0]
```

Output:

```
p
n
t
n
p
o
6
1
```

4.2.3 Updating Tuples

Tuples are immutable means that the tuple values cannot be updated or changed. However, the portions of existing tuples are added with a new tuple to create another tuple as the following example demonstrates −

Consider the following tuple,

```
t3= ('a', 'b', 'c', 'd', 'e')
```

No elements can be modified. If you try to modify, you will get an error.

```
t3[1]= 'B'
```

TypeError: 'tuple' object does not support item assignment

Instead of modifying an element in the tuple sequence, it is obvious to simply replace one tuple with another:

```
t3 = ('A',) + t3 [1:]
print t
```

Output:

```
('A', 'b', 'c', 'd', 'e')
```

Here, the first element ‘a’ is replaced with ‘A’. A new tuple is created with the value ‘A’ is combined with tuple t3 having index from 1 to the last element. The tuple value t3[0]=’a’ is replaced by ‘A’.

4.2.4 Delete Tuple Elements

It is impossible to delete a single element in the tuple. There is, of course, nothing wrong with putting together another tuple with the undesired elements discarded. To delete an entire tuple, the keyword **del** is used. For example:

```
t1 = (‘C’, ‘C++’, ‘python’, 1997, 2000);
print t1
del t1
print “After deleting : “
print t1
```

Output:

```
(‘C’, ‘C++’, ‘python’, 1997, 2000)
After deleting :
Traceback (most recent call last):
  File “main.py”, line 5, in
    print t1
NameError: name ‘t1’ is not defined
```

Program for updating and deleting tuples

```
t1 = (2, 3, 4, [5, 6])
print  t1
# we cannot change an element
# TypeError: ‘tuple’ object does not support item assignment
#my_tuple[1] = 6; creates error
# but item of mutable element can be changed
t1[3][0] = 7
print  t1
# tuples can be reassigned
t1 = (‘h’,’e’,’l’,’l’,’o’)
print  t1
# Concatenation
```

```
print  (1, 2, 3) + (4, 5, 6)
# Repetition operator
Print (“Repeat”,) * 3
#delete tuple
del t1
print  t1
```

Output:

```
(2, 3, 4, [5, 6])
(2, 3, 4, [7, 6])
(‘h’, ‘e’, ‘l’, ‘l’, ‘o’)
(1, 2, 3, 4, 5, 6)
(‘Repeat’, ‘Repeat’, ‘Repeat’)
Traceback (most recent call last):
File “<stdin>”, line 25, in <module>
print(t1)
NameError: name ‘t1’ is not defined
```

4.2.5 Tuple assignment

Tuple assignment makes it possible to create and assign values for more than one tuple in a single statement itself. For example,

```
x, y =  1, 2
print x
print y
```

Output:

```
1
2
```

In general to swap the values of two variables, a temporary variable is used. For example to swap x and y:

```
temp = x
x = y
y = temp
```

This solution is clumsy; the tuple assignment is more elegant.

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

In the expression, 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 side and the number of values on the right must be the same:

```
a, b = 1, 2
```

In this *a* is assigned with 1 and *b* is assigned with 2.

For the assignment,

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

This statement creates error, as

ValueError: too many values to unpack

The right side of the assignment statement can be any kind of sequence (string, list or tuple). For example, to split an email address into a user name and a domain, the split function can be used as follows.

```
mail_id = 'students@python.org'
uname, domain = mail_id.split('@')
print uname
print domain
```

Output:

```
students
python.org
```

In this, the split function is used to separate the value into two parts. The return value from the split function is a list with two elements; the first element is assigned to uname, the second is assigned to domain.

Program to illustrate tuple creation and assignment

```
# empty tuple
t1 = ()
print  t1
# tuple having integers
t2 = (1, 2, 3)
print  t2
# tuple with mixed datatypes
t3 = (1, "Hello", 2.4)
print  t3
# nested tuple
t4 = ("World", [8, 4, 6], (1, 2, 3))
```

```
print  t4
# tuple can be created without parentheses (called tuple packing)
t5 = 3, 4.6, “ABC”
print  t5
# tuple unpacking is also possible
#tuple assignment
x, y, z = t5
print  x
print  y
print  z
```

Output:

```
 ()
(1, 2, 3)
(1, ‘Hello’, 2.4)
(‘World’, [8, 4, 6], (1, 2, 3))
(3, 4.6, ‘ABC’)
3
4.6
ABC
```

4.2.6 Tuple Methods

In python methods for adding items and deleting items are not available. The methods available are *count* and *index*.

- count(*x*) method returns the number of occurrences of x
- index(x) method returns index of the first occurrence of the item x.

Program for count and index methods

```
t1 = (‘p’,’y’,’t’,’h’,’o’,’n’,’p’,’r’,’o’,’g’,’r’,’a’,’m’)
# Count
print(t1.count(‘p’))
# Index
print  t1.index(‘y’)
print  t1.index(‘h’)
```

Output:

```
2
1
3
```

4.2.7 Other Tuple Operations

There are some other tuple operations such as tuple membership test and iterating through a tuple. Tuple Membership Test operation can test if an item exists in a tuple or not, using the keyword `in and not in`. Iterating through a Tuple operation is performed using a `for` loop in which we can iterate through each item in a tuple.

Simple program to illustrate tuple operations

```
t1 = (‘p’,’y’,’t’,’h’,’o’,’n’)
# In operation
print(‘y’ in t1)
# Output: False
print(‘m’ in t1)
# Not in operation
print(‘h’ not in t1)
print(‘a’ not in t1)
for lang in (‘C’,’C++’):
    print(“Progrmming-languages”,lang)
```

Output:

```
True
False
False
True
Progrmming-languages C
Progrmming-languages C++
```

4.2.8 Tuples as return values

In general a function can return only one value, but if the return value is a tuple, then it is returning multiple values. For example, to divide two integers and compute the quotient and remainder, normally it is used to compute x/y and then x%y. But using python, it is better to compute them both at the same time. The following code explains this

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

Output:

```
(2, 1)
```

Here, the built-in function divmod is used which takes two arguments and returns a tuple of two values, the quotient and remainder. The result can be stored as a tuple as in previous program code. Or tuple assignment can be used to store the elements separately as in the following code.

```
quot, rem = divmod(7, 3)
print quot
print rem
```

Output:

```
2
1
```

One more example to explain tuples as return values. The built-in functions min and max are used to find the smallest and largest elements of a sequence. The function min_max computes both and returns a tuple of two values.

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

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

4.2.9 Built-in Functions with Tuple

Function	**Description**
all()	Return True if all elements of the tuple are true (or if the tuple is empty).
any()	Return True if any element of the tuple is true. If the tuple is empty, return False.
enumerate()	Return an enumerate object. It contains the index and value of all the items of tuple as pairs.
len()	Return the length (the number of items) in the tuple.
max()	Return the largest item in the tuple.
min()	Return the smallest item in the tuple
sorted()	Take elements in the tuple and return a new sorted list (does not sort the tuple itself).
sum()	Return the sum of all elements in the tuple.
tuple()	Convert an iterable (list, string, set, dictionary) to a tuple.

4.2.10 Variable-length argument tuples

The functions can take a variable number of arguments for implementation. A argument name that starts with (*) gathers the several arguments into a tuple. For example, *printall* function takes any number of arguments and prints them:

Example:

```
def printall (*args):# the function takes several args
print args
```

The argument name may be anything, but *args* is conventional. Here is the example to show how the function *printall* works:

```
printall(1, 2.0, ‘3’)
(1, 2.0, ‘3’)
```

The complement of gather is **scatter**. To pass a sequence of values to a function as multiple arguments, the * operator can be used. For example, consider the *divmod* function which takes exactly two arguments; doesn’t work with a tuple of variable length arguments:

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

Output:

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

But if you scatter the tuple, it works:

Instead of the above code, the code given below can be used for variable length arguments.

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

There are some other built-in functions which use variable-length argument tuples.

The *max* and *min* functions take any number of arguments:

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

Output:

```
3
```

The *sum* function does not take variable length arguments. It gives error.

```
sum(1,2,3)
```

Output:

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

4.2.11 Comparing tuples

With relational operators it is possible to work with tuples and other sequences. To compare two elements, Python starts by comparing the first element from each sequence. If the elements are equal, it goes on to the next elements, and so on, until it finds an element that is different. Subsequent elements are not considered (even if they are really big).

```
>>> (0, 1, 2) < (0, 3, 4)
True
```

The sort function also works in the same way. It sorts primarily by first element. But if there is a tie, it sorts by second element, and so on. This feature lends itself to a pattern called **DSU**. DSU stands for Decorate, Sort, Undecorate.

DSU:

Decorate a sequence by building a list of tuples with one or more sort keys preceding the elements from the sequence,

Sort the list of tuples, and **Undecorate** by extracting the sorted elements of the sequence.

For example, to sort a list of words from longest to shortest:

```
def sort_by_length (words):
t = []
for word in words:
t.append((len(word), word))
t.sort(reverse=True)
res = []
for length, word in t:
res.append(word)
return res
```

The first loop builds a list of tuples, where each tuple is a word preceded by its length. The sort function compares the first element, and its length first, and only considers the second element to break ties. The keyword argument reverse=True tells sort to go in decreasing order. The second loop traverses the list of tuples and builds a list of words in descending order of length. The following program explains the comparison function.

```
t1, t2 = (123, ‘xyz’), (456, ‘abc’)
print cmp (t1, t2)
print cmp (t2, t1)
t3 = t2 + (786, )
print cmp (t2, t3)
print cmp (t3, t2)
t3, t4= (‘hello’, 3) , (‘hello’, 3)
print cmp (t3, t4)
print cmp (t4,t3)
t5,t6 = (3, ‘hai’), (‘hai’,3)
print cmp (t5, t6)
print (t6, t5)
```

Output:

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

4.3 DICTIONARIES

Dictionary is an unordered collection of items. It is similar to a list, but in list elements can be accessed using index which must be an integer. In Dictionary we access values by looking up a **key** instead of an index. A key can be any string or number. For example, dictionaries can be used for things like phone books (pairing a name with a phone number), login pages (pairing an e-mail address with a username).

Each item in dictionary has a *key: value* pair and the list of items are enclosed inside curly braces {} separated by comma. The values can be of any data type and can repeat; keys must be of immutable types (string, number or tuple with immutable elements) and must be unique.

Dictionaries in Python are implemented using **hash table**. It is an array whose indexes are obtained using a hash function on the keys. A **hash function** takes a key value and returns hash value, an integer. This hash value is used in the dictionary to store and lookup key-value pairs. So keys in dictionary must be **hashable**.

The following code is a simple example which creates an empty dictionary.

```
# empty dictionary
my_dict = {}
print my_dict
```

Sample Output:

```
{}
```

The following dictionary uses integer as keys and string as values.

```
# dictionary with integer keys
my_dict = {1: 'apple', 2: 'ball'}
print my_dict
print my_dict[2]
```

Sample Output:

```
{1: ‘apple’, 2: ‘ball’}
ball
```

The following dictionary uses mixed keys. For item 1, both key and its corresponding value are string. In item 2, the key is an integer and the value is a list.

```
# dictionary with mixed keys
my_dict = {‘name’: ‘John’, 1: [2, 4, 3]}
print my_dict
print my_dict[‘name’]
print my_dict[1]
```

Sample Output:

```
{1: [2, 4, 3], ‘name’: ‘John’}
John
[2, 4, 3]
```

In the output, the order of the key-value pairs is not the same. In general, the order of items in dictionary is unpredictable. In the following example, using list, a mutable data type as key results in error message.

```
dic = { [1,2,3]:”abc”}
Traceback (most recent call last):
File “main.py”, line 1, in <module>
dic = { [1,2,3]:”abc”}
TypeError: unhashable type: ‘list’
```

Tuple, an immutable data type can be used as key, which is shown in following example.

```
my_dic = { (1,2,3):”abc”, 3.14:”abc”}
print my_dic
```

Sample Output:

```
{3.14: ‘abc’, (1, 2, 3): ‘abc’}
```

An exception will be raised when we try to access a key that not exist in dictionary. In the following example, accessing my_dict[2] results in an error, as the key 2 not exist in dictionary.

```
my_dict = {‘name’: ‘John’, 1: [2, 4, 3]}
print my_dict[2]
```

Sample Output:

```
Traceback (most recent call last):
File "main.py", line 2, in <module>
 print my_dict[2]
KeyError: 2
```

Dictionaries can also be created using the *dict()* function.

```
# using dict()
my_dict = dict({1:'apple', 2:'ball'})
print my_dict
```

Sample Output:

```
{1: 'apple', 2: 'ball'}
```

4.3.1 Built-in Dictionary Functions & Methods

Built-in methods or functions that are available with dictionary are tabulated below.

Function/Method	**Description**
len(dict)	Returns the length of dictionary which is equal to number of pairs in the dictionary.
cmp(dict1,dict2)	Compare items of two dictionaries
sorted(dict)	Returns sorted list of keys in dictionary
dict.clear()	Remove all items from dictionary
dict.copy()	Returns a shallow copy of dictionary
dict.fromkeys(*seq*[, *v*])	Return a new dictionary with keys from *seq* and value equal to *v*
dict.get(key)	Returns the value of key. If key does not exists, it returns None
dict.pop(key)	Remove the item with key and returns its value. *KeyError* occurs when key is not found
dict.popitem()	Remove and return an arbitary item (key, value). Raises *KeyError* if the dictionary is empty.
dict.items()	Returns a list of *dict*'s (key, value) tuple pairs
dict.keys()	Returns list of dictionary dict's keys
dict1.update(dict2)	Update the dictionary *dict1* with the key/value pairs from *dict2*, overwriting existing keys.

4.3.2 Access, update, and add elements in dictionary

Key can be used either inside square brackets or with the get() method. The difference while using get() is that it returns None instead of KeyError, if the key is not found. Dictionary is mutable.

So we can add new items or change the value of existing items. If the key is present, its value gets updated. Else a new key:value pair is added to dictionary.

```
my_dict={‘name’:’Ram’,’age’:21}
print my_dict # display all items
print my_dict.get(‘name’) # Retrieves the value of name key
my_dict[‘age’]=23 # update value
print my_dict
my_dict[‘dept’]=’CSE’ # add item
print my_dict
```

Sample Output:

```
{‘age’: 21, ‘name’: ‘Ram’}
{‘age’: 23, ‘name’: ‘Ram’}
{‘dept’: ‘CSE’, ‘age’: 23, ‘name’: ‘Ram’}
```

4.3.3 Delete or remove elements from a dictionary

A particular item in a dictionary can be removed by using the method *pop()*. This method removes as item with the provided key and returns the value. The method, *popitem()* can be used to remove and return an arbitrary item (key, value) from the dictionary. All the items can be removed at once using the *clear()* method.

```
squares={1:1,2:4,3:9,4:16,5:25}
print(squares.pop(3)) # remove a particular item
print squares
print (squares.popitem()) # remove an arbitrary item
print squares
del squares[5] # delete a particular item
squares.clear() # remove all items
print squares
```

Sample Output:

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

We can also use the *del* keyword to remove individual items or the entire dictionary itself. If we try to access the deleted dictionary, it will raise an Error.

```
del squares # delete the dictionary itself
print squares #throws error
Traceback (most recent call last):
File “main.
py”, line 11, in <module>
print squares NameError: name ‘squares’ is not defined
```

4.3.4 Sorting a Dictionary

The items in dictionary can be sorted using *sorted()* function. In the following example, *fromkeys()* function is used to create a dictionary from sequence of values. The value 0 is assigned for all keys. Each item is accessed iteratively using for loop that iterate though each key in a dictionary.

```
marks={}.fromkeys([‘Math’,’English’,’Science’],0)
print marks
for item in marks.items():
        print item
print list(sorted(marks.keys()))
```

Sample Output:

```
{‘Maths’: 0, ‘Science’: 0, ‘English’: 0}
(‘Maths’, 0)
(‘Science’, 0)
(‘English’, 0)
[‘English’, ‘Maths’, ‘Science’]
```

4.3.5 Iterating Through a Dictionary

Using a `for` loop we can iterate though each key in a dictionary.

```
squares={1:1,2:4,3:9,4:16,5:25}
for i in squares:
        print(squares[i])
```

Sample Output:

```
1
4
9
16
25
```

4.3.6 Reverse Lookup

Lookup is the process of finding the corresponding value for the given key from dictionary. It's easy to find the value given a key to a python dictionary.

```
value=dict[key]
```

Whereas, **reverse lookup** is the process of finding the key for a given value. There is no direct method to handle reverse lookup. The following function takes a value and returns the first key that map to that value.

```
def get_Value(dic,value):
    for name in dic:
        if dic[name] == value:
            return name
    raise ValueError
squares={1:1,2:4,3:9,4:16,5:25}
print get_Value(squares,4) # successful reverse lookup
```

Sample Output:

```
2
```

In this example, *raise* keyword is used to raise/activate an exception. *ValueError* indicates that there is something wrong with value of parameter. On unsuccessful reverse lookup, when the value is not in the dictionary, the exception *ValueError* is raised. Unsuccessful reverse lookup result in following error.

```
print get_Value(squares,6) # unsuccessful reverse lookup
Traceback (most recent call last):
File "main.py", line 7, in <module>
print get_Value(squares,6)
File "main.py", line 5, in get_Value
raise ValueError
ValueError
```

4.3.7 Inverting a Dictionary

A dictionary can be inverted with list values. For example, if you were given a dictionary that maps from child to parent, you might want to invert it; that is, create a dictionary that maps from parent to children. Since there might be several children with the same parent each value in the inverted dictionary should be a list of children.

```
def invert_dict_nonunique(d):
    newdict = {}
    for k, v in d.iteritems():
        newdict.setdefault(v, []).append(k)
    return newdict
d = {'child1': 'parent1',
    'child2': 'parent1',
    'child3': 'parent2',
    'child4': 'parent2',
    }
print invert_dict_nonunique(d)
```

Sample Output:

```
{'parent2': ['child3', 'child4'], 'parent1': ['child1', 'child2']}
```

In this example the loop iterates through dictionary items where *k* represents key and *v* represents value. The *setdefault()* method will set *newdict[v]=[]* and append the key value to list..

As we mentioned earlier, the keys in dictionaries have to hashable. It works correctly only when keys are immutable. For example, if key is a list and to store a key-value pair Python will hash the key and store it in corresponding location. If that key is modified, it would be hashed to different location. In this case, we will have two entries for the same key or might be failed to locate a key. Either way, the dictionary wouldn't work correctly. Since lists and dictionaries are mutable, they can't be used as keys, but they can be used as values.

4.3.8 Memoization (Memos)

Memoization effectively refers to remembering results of method calls based on the method inputs and then returning the remembered result rather than computing the result again.

For example, consider the recursive version to calculate the Fibonacci numbers. The following code to compute Fibonacci series has an exponential runtime behavior.

```
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)
```

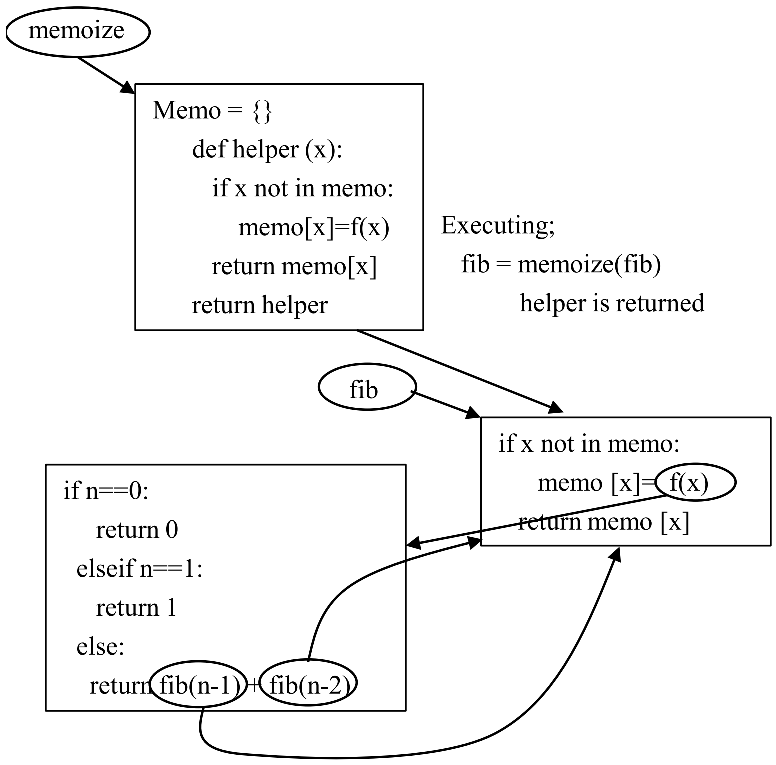
The runtime behavior of this recursive version can be improved by adding a dictionary to memorize previously calculated values of the function.

```
def memoize(f):
    memo = {}
    def helper(x):
        if x not in memo:
            memo[x] = f(x)
        return memo[x]
    return helper
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)
fib = memoize(fib)
print(fib(6)) # output the 6th number in Fibonacci series (series starts from 0th position)
```

Sample Output:

```
8
```

memoize() takes a function as an argument. The function *memorize()* uses a dictionary "memo" to store the function results. Though the variable "memo" as well as the function "f" are local to *memoize*, they are captured by a closure through the helper function which is returned as a reference by *memoize()*. So, the call *memoize(fib)* returns a reference to the *helper()* which is doing what *fib()* would do on its own plus a wrapper which saves the calculated results. For an integer 'n' *fib(n)* will only be called, if n is not in the memo dictionary. If it is in it, we can output memo[n] as the result of *fib(n)*.



After having executed fib = memoize(fib) fib points to the body of the helper function, which had been returned by memoize. The decorated Fibonacci function is called in the return statement return fib(n-1) + fib(n-2), this means the code of the helper function which had been returned by memorize.

4.4 ILLUSTRATIVE PROGRAMS

4.4.1 Python program to sort a list of elements using the selection sort algorithm

Selection sort is a simple sorting algorithm. It is an in-place comparison-based algorithm in which the list is divided into two parts: the sorted part at the left end and the unsorted part at the right end. Initially, the sorted part is empty and the unsorted part is the entire list.

Selection sort algorithm starts by comparing first two elements of an array and swapping if necessary, i.e., if you want to sort the elements of array in ascending order and if the first element is greater than second then, you need to swap the elements but, if the first element is smaller than second, leave the elements as it is. Then, again first element and third element are compared and swapped if necessary. This process goes on until first and last element of an array is compared. This completes the first step of selection sort. The working of selection sort algorithm is shown in Figure.4.3.

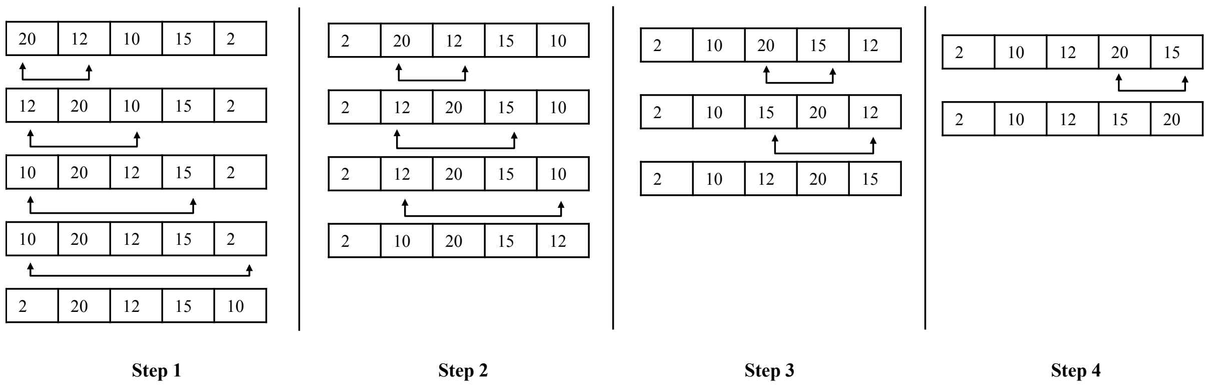


Figure 4.3. Selection Sort

This algorithm is not suitable for large data sets as its average and worst case complexities are of O(n^2), where *n* is the number of items.

```
arr=[]
n=input('Enter number of elements')
for i in range(0,n):
        x=int(input('Enter number'))
        arr.insert(i,x)
        i+=1
for i in range(len(arr)):
   for j in range(i, len(arr)):
        if(arr[i] > arr[j]):
            arr[i], arr[j] = arr[j], arr[i]
print 'Sorted List:', arr
```

Sample input/output:

```
Enter no5
enter no.12
 enter no. 2
enter no.23
enter no. 4
enter no.5
Sorted List:
[2, 4, 5, 12, 23]
```

4.4.2 Python program to sort a list of elements using the insertion sort algorithm

Insertion sort is a simple sorting algorithm that builds the final sorted array (or list) one item at a time. Here, a sub-list is maintained which is always sorted. The array is searched sequentially and unsorted items are moved and inserted into the sorted sub-list (in the same array). This algorithm is suitable for small data sets. But it is much less efficient on large lists than more advanced algorithms such as quicksort, heapsort, or merge sort. The worst case complexity of the algorithm is of O(n^2), where *n* is the number of items.

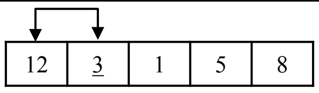
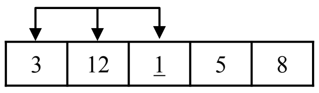
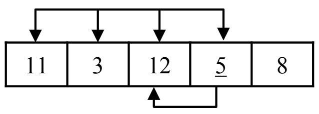
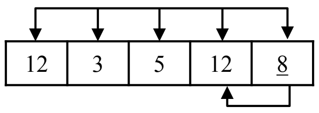
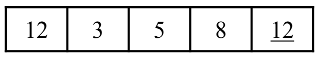
Step 1		Checking second element of array with element before it and inserting it in proper position. In this case, 3 is inserted in position of 12.
Step 2		Checking third element of array with element before it and inserting it in proper position. In this case, 1 is inserted in position of 3.
Step 3		Checking fourth element of array with element before it and inserting it in proper position. In this case, 5 is inserted in position of 12.
Step 4		Checking fifth element of array with element before it and inserting it in proper position. In this case, 8 is inserted in position of 12.
		Sorted Array in Ascending Order

Figure 4.4. Insertion Sort

```
def insertionsort(list):
    for i in range(1,len(list)):
        temp=list[i]
```

```
            j=i-1
            while temp<+list[j] and j>=0:
                list[j+1]=list[j]
                j=j-1
            list[j+1]=temp
        return list
    arr=[]
    n=input('Enter number of elements')
    for i in range(0,n):
            x=int(input('Enter number'))
            arr.insert(i,x)
            i+=1
    print insertionsort(arr)
```

Sample input/output:

```
Enter number of elements5
Enter number12
Enter number23
Enter number4
Enter number16
Enter number34
[4, 12, 16, 23, 34]
```

4.4.3 Python program to sort a list of elements using the merge sort algorithm

Merge sort is a sorting technique based on divide and conquer technique. It first divides the array into equal halves and then combines them in a sorted manner. The basic steps involved in merge sort algorithm are as follows: Given an array *A*.

1. Divide

If q is the half-way point between p and r, then we can split the subarray $A[p..r]$ into two arrays $A[p..q]$ and $A[q+1, r]$.

2. Conquer

In the conquer step, we try to sort both the subarrays $A[p..q]$ and $A[q+1, r]$. If we haven't yet reached the base case, we again divide both these subarrays and try to sort them.

3. Combine

When the conquer step reaches the base step and we get two sorted subarrays *A*[*p*..*q*] and *A*[*q*+1, *r*] for array *A*[*p*..*r*], we combine the results by creating a sorted array *A*[*p*..*r*] from two sorted subarrays *A*[*p*..*q*] and *A*[*q*+1, *r*].

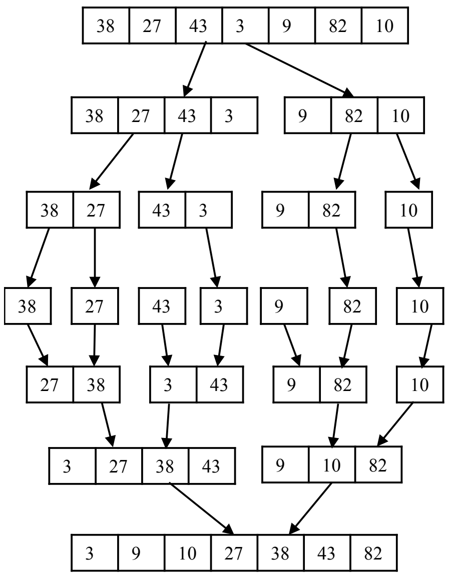


Figure 4.5. Merge Sort

As shown in Figure. 4.5, the merge sort algorithm recursively divides the array into halves until we reach the base case of array with 1 element. After that, the merge function picks up the sorted sub-arrays and merges them to gradually sort the entire array. The worst case complexity of the algorithm is O(n log n), where *n* is the number of items.

```
def merge_sort(sequence):
    if len(sequence) < 2:
        return sequence
    m = len(sequence) / 2
    return merge(merge_sort(sequence[:m]), merge_sort(sequence[m:]))
def merge(left, right):
    result = []
    i = j = 0
    while i < len(left) and j < len(right):
        if left[i] < right[j]:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1
    result += left[i:]
    result += right[j:]
    return result
print merge_sort([5, 2, 6, 8, 5, 8, 1])
```

Sample output:

```
[1, 2, 5, 5, 6, 8, 8]
```

4.4.4 Python program to sort a list of elements using the Quick sort algorithm

Like Merge Sort, Quick Sort is a Divide and Conquer algorithm. It picks an element as pivot and partitions the given array around the picked pivot. The pivot element can be selected using following different ways.

(1) Always pick first element as pivot.

(2) Always pick last element as pivot.

(3) Pick a random element as pivot.

(4) Pick median as pivot.

The runtime of the algorithm varies based on the pivot selected. The basic idea behind this algorithm is as follows.

1. Pick one element in the array, which will be the *pivot*.
2. Make one pass through the array, called a *partition* step, re-arranging the entries so that:
   i) the pivot is in its proper place.
   ii) entries smaller than the pivot are to the left of the pivot.
   iii) entries larger than the pivot are to its right.
3. Recursively apply quick sort to the part of the array that is to the left of the pivot, and to the right part of the array.

The steps involved in quick sort algorithm are listed below and can be understand easily using the example shown in Figure.4.6.

Step 1 − Choose the highest index value has pivot

Step 2 − Take two variables to point left and right of the list excluding pivot

Step 3 − left points to the low index

Step 4 − right points to the high

Step 5 − while value at left is less than pivot move right

Step 6 − while value at right is greater than pivot move left

Step 7 − if both step 5 and step 6 does not match swap left and right

Step 8 − if left ≥ right, the point where they met is new pivot

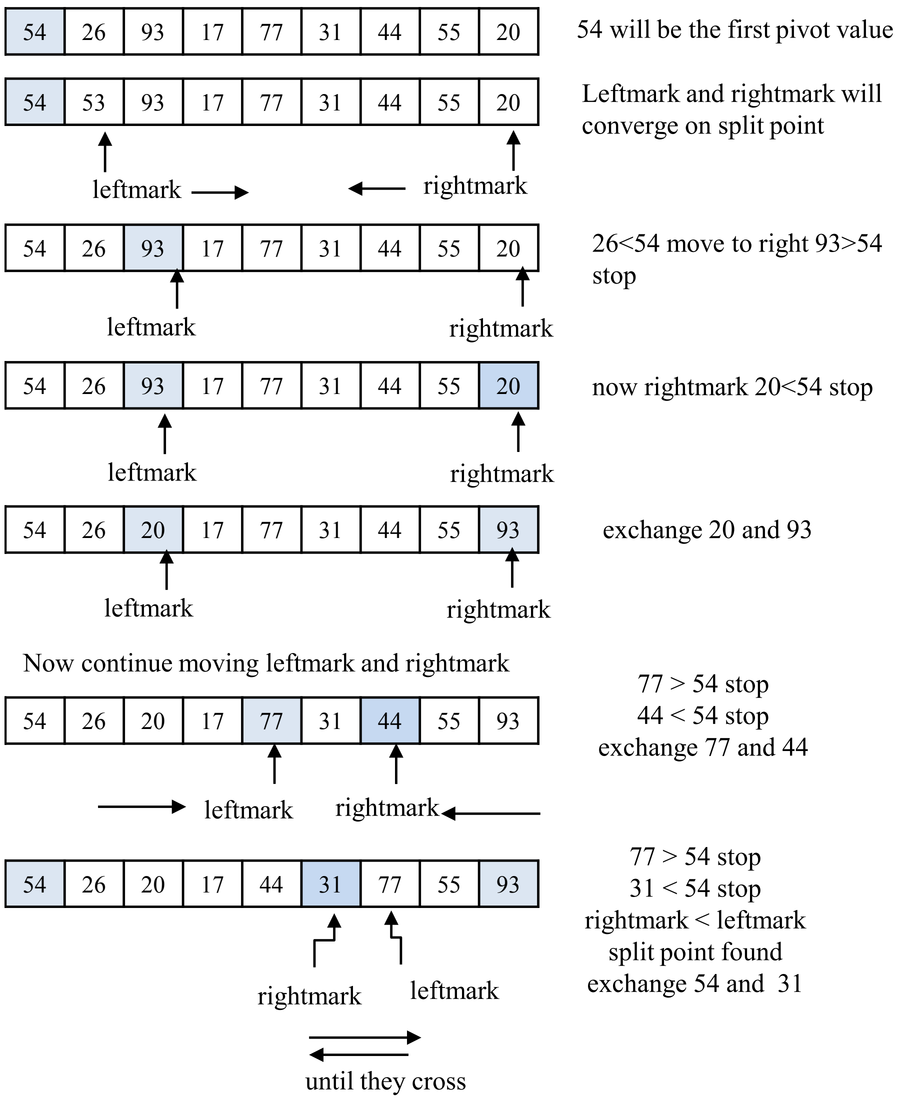


Figure 4.6. Quick Sort

```
# divide part of the algorithm
def partition(myList, start, end):
    pivot = myList[start]
    left = start+1
    right = end
    done = False
    while not done:
            while left <= right and myList[left] <= pivot:
                left = left + 1
            while myList[right] >= pivot and right >=left:
                right = right -1
            if right < left:
                done= True
            else:
                # swap places
                temp=myList[left]
                myList[left]=myList[right]
                myList[right]=temp
        # swap start with myList[right]
        temp=myList[start]
        myList[start]=myList[right]
        myList[right]=temp
        return right
    # conquer part of the quicksort routine
    def quicksort(myList, start, end):
        if start < end:
            # partition the list
            pivot = partition(myList, start, end)
            # sort both halves
            quicksort(myList, start, pivot-1)
            quicksort(myList, pivot+1, end)
        return myList
    List = [54,26,93,17,77,31,44,55,20]
    print quicksort(List)
```

Sample Output:

```
[17, 20, 26, 31, 44, 54, 55, 77, 93]
```

4.4.5 Write a Python program to create a histogram from a given list of integers.

```
def histogram( items ):
    for n in items:
        output = ''
        times = n
        while( times > 0 ):
          output += '*'
          times = times - 1
        print(output)
    histogram([2, 3, 6, 5])
```

Sample Output:

```
**
***
******
*****
```

TWO MARKS QUESTION & ANSWER

1. **How will you access the elements of a list in Python?**

   To access the element(s) of a list, subscript operator [ ] (also known as slicing operator) is used. Index within [ ] indicates the position of the particular element in the list and it must be an integer expression. For eg, in a list stulist = [‘Ram’, ‘Chennai’, 2017], stulist[1] returns Chennai as its output.

2. **What is the use of + and * operators in Python lists?**

   In Python lists, + represents concatenation operation and * represents repetition operation.

Eg:

```
a=[10, 20, 30]
b=[40, 50]
c=a+b
print c
d=c*2
print d
```

Output:

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

3. **Define List slice.**

   A part of a list is called list slice. The operator [m:n] returns the part of the list from mth index to nth index, including the element at mth index but excluding the element at nth index.

   - If the first index is omitted, the slice starts at the beginning of the string.
   - If the second index is omitted, the slice goes to the end of the string.
   - If the first index is greater than or equals to the second, the slice is an empty string.
   - If both indices are omitted, the slice is a given string itself.

4. **Mention some of the methods of Python lists.**

   - append() - to add element to the end of specified list
   - count() - to count the number of occurrences of an element in a specified list
   - extend() – to append the contents of secondlist to the firstlist
   - insert() – to Insert the given element at the given index in a specified list

5. **Can we change the elements of Python list?**

   Yes, the elements of Python list can be replaced, inserted and removed (List is a mutable data structure). A slice operator on the left side of an assignment operation can update single or multiple elements of a list. New elements can be added to the list using append() method.

6. **What do you mean by list cloning?**

   In lists, cloning operation creates a copy of an existing list so that changes made in one copy of list will not affect another. The copy contains the same elements as the original.

   How will you convert a string to a list in python?

   list(s) − Converts a string s to a list.

Eg:

```
s='abc'
print s
print list(s)
```

Output:

```
abc
['a', 'b', 'c']
```

7. **What is the difference between del() and remove() methods of list?**

   To remove a list element, we can use either the del statement if we know exactly which element(s) we are deleting or the remove() method if we do not know.

8. **How do you remove duplicates from a list?**

Steps:

(a) sort the list.

(b) scan the list from the end.

(c) while scanning from right-to-left, delete all the duplicate elements from the list

9. **Differentiate append() and extend() methods.**

   Both append() and extend() methods are the methods of list. These methods a re 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.

10. **Define tuple.**

    A tuple is a collection of values of different types. Unlike lists, tuple values are indexed by integers. The important difference is that tuples are immutable.

Example:

```
t1 = (‘a’, ‘b’, ‘c’, ‘d’, ‘e’)
```

11. List any two tuple methods.

- count(x) method returns the number of occurrences of x
- index(x) method returns index of the first occurrence of the item x.

12. Define Dictionary.

Dictionary is an unordered collection of items. In Dictionary we access values by looking up a key instead of an index. A key can be any string or number. Dictionaries in Python are implemented using hash table.

Example:

```
my_dict = {1: ‘apple’, 2: ‘ball’}
```

13. Compare lookup and reverse lookup.

Lookup is the process of finding the corresponding value for the given key from dictionary. It’s easy to find the value given a key to a python dictionary.

```
value=dict[key]
```

Whereas, reverse lookup is the process of finding the key for a given value.

14. Define Memoization.

Memoization effectively refers to remembering results of method calls based on the method inputs and then returning the remembered result rather than computing the result again.

15. How will you access the elements of a list in Python?

To access the element(s) of a list, subscript operator [ ] (also known as slicing operator) is used. Index within [ ] indicates the position of the particular element in the list and it must be an integer expression. For eg, in a list stulist = ['Ram', 'Chennai', 2017], stulist[1] returns Chennai as its output.

16. What is the use of + and * operators in Python lists?

In Python lists, + represents concatenation operation and * represents repetition operation.

Eg:

```
a=[10, 20, 30]
b=[40, 50]
c=a+b
print c
d=c*2
```

```
print d
```

Output:

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

17. Define List slice.

A part of a list is called list slice. The operator [m:n] returns the part of the list from mth index to nth index, including the element at mth index but excluding the element at nth index.

- If the first index is omitted, the slice starts at the beginning of the string.
- If the second index is omitted, the slice goes to the end of the string.
- If the first index is greater than or equals to the second, the slice is an empty string.
- If both indices are omitted, the slice is a given string itself.

18. Mention some of the methods of Python lists.

- append() - to add element to the end of specified list
- count() - to count the number of occurrences of an element in a specified list
- extend() – to append the contents of secondlist to the firstlist
- insert() – to Insert the given element at the given index in a specified list

19. Can we change the elements of Python list?

Yes, the elements of Python list can be replaced, inserted and removed (List is a mutable data structure). A slice operator on the left side of an assignment operation can update single or multiple elements of a list. New elements can be added to the list using append() method.

20. What do you mean by list cloning?

In lists, cloning operation creates a copy of an existing list so that changes made in one copy of list will not affect another. The copy contains the same elements as the original.

How will you convert a string to a list in python?

list(s) − Converts a string s to a list.

Eg:

```
s='abc'
print s
print list(s)
Output:
abc
['a', 'b', 'c']
```

21. What is the difference between del() and remove() methods of list?

To remove a list element, we can use either the del statement if we know exactly which element(s) we are deleting or the remove() method if we do not know.

22. How do you remove duplicates from a list?

Steps:

(a) sort the list.

(b) scan the list from the end.

(c) while scanning from right-to-left, delete all the duplicate elements from the list

23. Differentiate append() and extend() methods.

Both append() and extend() methods are the methods of list. These methods a re 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.

24. Define tuple.

A tuple is a collection of values of different types. Unlike lists, tuple values are indexed by integers. The important difference is that tuples are immutable.

Example:

```
t1 = ('a', 'b', 'c', 'd', 'e')
```

25. List any two tuple methods.

- count(x) method returns the number of occurrences of x
- index(x) method returns index of the first occurrence of the item x.

26. Define Dictionary.

Dictionary is an unordered collection of items. In Dictionary we access values by looking up a key instead of an index. A key can be any string or number. Dictionaries in Python are implemented using hash table.

Example:

```
my_dict = {1: 'apple', 2: 'ball'}
```

27. Compare lookup and reverse lookup.

Lookup is the process of finding the corresponding value for the given key from dictionary. It's easy to find the value given a key to a python dictionary.

```
value=dict[key]
```

Whereas, reverse lookup is the process of finding the key for a given value.

28. Define Memoization.

Memoization effectively refers to remembering results of method calls based on the method inputs and then returning the remembered result rather than computing the result again.

29. What is a List?

A list is a sequence of any type of values and can be created as a set of comma-separated values within square brackets. The values in a list are called elements or items. A list within another list is called nested list.

30. What is meant by mutable data structure?

The list is a mutable data structure where we can make changes over the list. This means that its elements can be replaced, inserted and removed. A slice operator on the left side of an assignment operation can update single or multiple elements of a list. New elements can be added to the list using append() method.

31. How to create copy of a list?

In lists, cloning operation can be used to create a copy of an existing list so that changes made in one copy of list will not affect another. The copy contains the same elements as the original.

32. What is list Comprehension and what are its components?

Comprehensions are constructs that allow sequences to be built from other sequences. It provides a concise way to create lists. Python 2.0 introduced list comprehensions and Python 3.0 comes with dictionary and set comprehensions.

A list comprehension consists of the following parts:

- An Input Sequence.
- Variable representing members of the input sequence.
- An Optional Predicate expression.
- An Output Expression producing elements of the output list from members of the Input Sequence that satisfy the predicate.

33. List the advantages of tuple over list.

- Tuples generally used for heterogeneous (different) datatypes and list for homogeneous (similar) datatypes.
- Tuples are immutable, so iterating through tuple is faster than with list. There is a slight performance enhancement through list.
- Tuple elements can be used as key for a dictionary. With list, this is not possible.
- If you have data that doesn't change, implementing it as tuple will guarantee that it remains write-protected.

34. How do you create a dictionary which can preserve the order of pairs?

We know that regular Python dictionaries iterate over <key, value> pairs in an arbitrary order, hence they do not preserve the insertion order of <key, value> pairs. Python 2.7. introduced a new “OrderDict” class in the “collections” module and it provides the same interface like the general dictionaries but it traverse through keys and values in an ordered manner depending on when a key was first inserted.

Eg:

```
from collections import OrderedDict

d=OrderDict([('Company-id':1),('Company-Name':'Intellipaat')])

d.items() # displays the output as: [('Company-id':1),('Company-Name':'Intellipaat')]
```

35. 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.

36. How many kinds of sequences are supported by Python? What are they?

Python supports 7 sequence types. They are str, list, tuple, unicode, bytearray, xrange, and buffer. where xrange is deprecated in python 3.5.X.