Unit-2: Advanced Programing with Python

Chpater-1 : Linear List Manipulation

Learning Objectives

At the end of this chapter the student will be able to:

- Understand data structures
- Understand sequential memory allocation
- Learn basic list operations -
  - Traversal in a list
  - Insertion in a sorted list
  - Deletion of an element from a list
- Learn Searching Techniques in a list-
  - Linear Search
  - Binary Search
- Learn Sorting a list
  - Selection Sort
  - Bubble Sort
  - Insertion Sort

Data Structures

A data structure is a group of data which can be processed as a single unit. This group of data may be of similar or dissimilar data types. Data Structures are very useful while programming because they allow processing of the entire group of data as a single unit. This makes managing of data simpler. The simplest form of data structure is an array which combines finite data of same data type.

Data structures are of two types: Linear and Non - Linear. In a linear data structure, the elements are stored in a sequential order. On the other hand, in a non linear data structure no sequential order is followed. It is a sort of multilevel data structure. Arrays, lists, stacks, queues, linked lists etc. are examples of linear data structure while tree, graph etc. is a non - linear data structure.

List is one of the simplest and most important data structures in Python. In class XI, you have studied that a list is a sequence of values of any data type. These ordered set of values are called elements or members of a list and are enclosed in square brackets[ ]. To identify a value of a list, we use index. In this chapter we will study how to traverse, edit and sort the list.

Implementation of List in memory

In any programming language, an array is defined as a set of contiguous data of similar data type. In most of the old languages, the length of the array is fixed at the time of declaration, although now recent languages provide the facility of variable length arrays. Python lists are actually arrays of variable length. The elements of a list are of heterogeneous types which means they are of different data types. Hence [4,"Good", 8.45, 'k'] is a valid list. This is possible because in Python language, technically pointers., and not objects are stored in the form of an array. So a list in Python is an array that contains elements (pointers to objects) of a specific size only and this is a common feature of all dynamically typed languages. For implementation of a list, a contiguous array of references to other objects is used. Python keeps a pointer to this array and the array's length is stored in a list head structure. This makes indexing of a list independent of the size of the list or the value of the index. When items are appended or inserted, the array of references is resized.

Sequential Memory Allocation

As you have studied before, a list is allocated memory in sequential manner. This means that the elements of the list are stored in memory in sequence of their declaration. So if you want to view the fourth element of the list, you have to first traverse through first three elements of the list. This is called sequential allocation of memory.

The list is allocated memory in accordance with its members and their types. Also if memory is being shared by two or more members of the list, then memory consumption will be different from the situation where no sharing is involved. In Python, the length of the lists is not fixed as elements can be dynamically added and removed from the lists.

Secondly memory consumption of the list is referenced by a pointer, which will take 4 bytes in 32 bit mode and 8 bytes in 64 bit mode. Here mode refers to the word size of the processor (Remember Unit- I of class XI). Hence the size of list will be 4 times the number of objects in the list in the 32 bit mode and 8 times the number of objects in the list in 64 bit mode. Other than these every list has a fixed header overhead and some other over allocations involved for all Python lists.

Considering all the given factors, calculation of address of individual elements is beyond the scope of this book.

List Operations

Traversal, insertion and deletion are three main operations of any list in Python. Let us study all these operations in detail.

> Please note that in all algorithms and programs that follow, DATA_LIST is the list containing size elements. U is the upper bound or the maximum index of the list and is equal to size-1. Also for convenience, we also assume that DATA_LIST contains only integer elements.

Traversal in a List

Traversal means to move in a list in sequential manner starting from first element to the last element. The algorithm and code for traversal in the list is as follows:

Algorithm

1. ctr=0
2. Repeat steps 3 through 4 until ctr>U
3. print DATA_LIST[ctr]
4. ctr=ctr+1
   #End of repeat
5. END

Program Code

```
def traversal(DATA_LIST):
    for x in DATA_LIST:
        print x
    print
```

In the above code, the for loop runs till the time all elements of the list are displayed.

Insertion of an element in a sorted list

If insertion is done in an unsorted list, the new element is inserted at the end of the list, using append(). You have studied about this in class XI. But if the list is sorted, then the new element has to be inserted at an appropriate position so that it falls in order with the other elements of the list. To insert the new element at the proper position, the rest of the elements have to be shifted as shown in the figure:

Original List		Space created for the new element	List after insertion
2	Element to be inserted	2	2
6	12	6	6
7		7	7
10		10	10
14			12
15		14	14
16		15	15
19		16	16
		19	19

Fig: Insertion in a sorted list

If the list is sorted in ascending order, then the appropriate position for the new element, say ELEM has to be determined first. If the appropriate position is i+1, then DATA_LIST[i]<=ELEM<=DATA_LIST[i+1].

Algorithm

1. Set ctr=0, U=size-1
2. if DATA_LIST[ctr]>ELEM then

   position=1

   else
3. { Repeat steps 5 and 6 until ctr>=U
4. if DATA_LIST[ctr]<=ELEM and ELEM<=DATA_LIST[ctr+1] then

   { position=ctr+1

   break;

   }
5. ctr=ctr+1

   #End of repeat
6. if ctr=U then

   position= U+1

   }
7. ctr=U
8. while ctr>=position perform steps 10 through 11
9. { DATA_LIST[ctr+1]=DATA_LIST[ctr]
10. ctr=ctr-1

    }
11. DATA_LIST[position]=ELEM
12. END

The above algorithm can be implemented in Python to develop a program to insert a new element in the sorted list. But Python provides a more efficient and simple way to add a new element into the sorted list. It is through the bisect module which implements the given algorithm in Python for inserting elements into the list while maintaining list in sorted order. It is found to be more efficient than repeatedly or explicitly sorting a list.

Consider the following program which generates a random number and inserts them into a sorted list.

```
#Generate random numbers and inserts them into a list sorted in ascending order.
import bisect
import random
random.seed(1)
print 'New Element---Location---Contents of the list'
print
 DATA_LIST = []
for i in range(1, 15):
     var = random.randint(1, 100)
    loc = bisect.bisect(DATA_LIST, var)
    bisect.insort(DATA_LIST, var)
    print '%3d      %3d' % (var, loc), DATA_LIST
```

The above program uses the random module and bisect module. The seed method of the random module uses a constant seed to ensure that the same pseudo random numbers are used each time a loop is run. The randint method of this module generates a random number between 1 and 100.

The bisect method of the bisect module used in the program given above gives the appropriate location of the new element and the insort method inserts the new element in the proper position.

The output of the above program will be

```
New Element--- Location---Contents of the list
    19          0       [19]
    4           0       [4, 19]
    54          2       [4,19,54]
    26          2       [4,19,26,54]
    80          4       [4,19,26,54,80]
    32          2       [4,19,26,32,54,80]
    75          5       [4,19,26,32,54,75,80]
    49          4       [4,19,26,32,49,54,75,80]
    2           0       [2,4,19,26,32,49,54,75,80]
```

In the above output, the first column of the output shows the new random number. The second column shows the position or location where the number will be inserted into the list. The last column shows the content of the list.

Deletion of an element from the sorted array

To delete the element from the sorted list, the element to be deleted has to be first searched for in the array using either linear search or binary search algorithm (discussed in the next section of this chapter). If the search is successful, the element is removed from the list and the rest of the elements of the list are shifted such that the order of the array is maintained. Since the elements are shifted upwards, the free space is available at the end of the array.

Algorithm

1. ctr=position # position is the position of the element to be deleted
2. Repeat steps 3 and 4 until ctr<=U
3. DATA_LIST[ctr]=DATA_LIST[ctr+1]
4. ctr=ctr+1

   #end of repeat

Python code to delete an element at the position pos, from a sorted list

```
def delete_element(DATA_LIST , pos):
ctr=pos
while(ctr<=U):          #U= size of list-1
    DATA_LIST[ctr]=DATA_LIST[ctr+1]
    ctr=ctr+1
```

Searching Techniques

There are many searching algorithms. We shall be discussing two of them - Linear Search and Binary Search.

Linear search

In linear search, the search element is compared with each element of the list, starting from the beginning of the list. This continues till either the element has been found or you have reached to the end of the list. That is why this type of searching technique is also called linear search.

The algorithm given below searches for an element, ELEM in the list named DATA_LIST containing size number of elements.

Algorithm

1. Set ctr=0 , U= size -1 # size is the size of the list L
2. Repeat steps 3 through 4 until ctr>U
3. if DATA_LIST[ctr]==ELEM then
   { print " Element found at "
   print ctr +1
   break
   }
4. ctr= ctr+1 # repeat ends
5. if ctr>U then
   print "Element not found"
6. END

Python Program to search an element ELEM from the list called DATA_LIST

```
def linear_search(DATA_LIST, ELEM):
    flag=0
for ctr in DATA_LIST:
    if DATA_LIST[ctr]==ELEM :
        print " Element found at "
        print ctr +1
        flag=1
        break
 if flag==0:
    print "Search not successful----Element not found"
```

Linear Searching technique is simple to use but has few drawbacks. If the element to be searched is towards the end of the array, the searching process becomes very time consuming. This is because the algorithm searches for the element in a sequential manner starting form the first element. This drawback is overcome to a large extent in the next sorting technique i.e. binary search.

Binary search

This searching technique reduces the number of comparisons and hence saves on processing time. For binary search, the condition that has to be fulfilled is that the array should be sorted in either ascending or descending order.

To search for an element, ELEM in a list that is sorted in ascending order, the ELEM is first compared with the middle element. If ELEM is greater than the middle element, latter part of the list is searched. So, this latter part becomes the new sub-array or the new segment of the array to be scanned further. On the contrary, if ELEM is smaller than the middle element, former part of the list becomes the new segment. For the first stage, the segment contains the entire array. In the next stage the segment is half of the list, in next stage it becomes one-fourth of the entire list and so on. So, the number of comparisons to be made, in this case are reduced proportionately, thereby decreasing the time spent in searching the element.

Let us assume that the element, ELEM has to be searched in a list that is sorted in ascending order. ELEM is first compared with the middle element of the list. If ELEM is greater than the middle element of the list, the second half of the list becomes the new segment to be searched. If the ELEM is less than the middle element, then the first half of the list is scanned for the ELEM. The process is repeated till either the element is found or we are left with just one element in the list to be checked and the ELEM is still not found.

Algorithm

In the algorithm given below, low and high are upper bound and lower bound respectively of DATA_LIST.

1. Set low=0 , high=size-1
2. Repeat steps 3 through 6 until low<high
3. mid=(int) (low+high)/2
4. if DATA_LIST[mid]==ELEM then

   {

   print "Element found at"

   print mid

   break;

   }
5. if DATA_LIST[mid]<ELEM then

   low=mid + 1
6. if DATA_LIST[mid]>ELEM then

```
        high=mid-1                    #End of Repeat
7.  if low >= high
        Print "ELEMENT NOT FOUND"
8.  END
```

Python Program

```
def binary_search(DATA_LIST, ELEM, low, high):
    low=0
    high=len(DATA_LIST)
while(low<high):
    mid=(int)(low+high/2)
    if DATA_LIST[mid]==ELEM:
        print "Element found at"
        print mid
        break
    if DATA_LIST[mid]<ELEM:
        low=mid+1
    if DATA_LIST[mid]>ELEM:
        high=mid-1
    if low >= high
        print "ELEMENT NOT FOUND"
```

Sorting a list

Sorting is to arrange the list in an ascending or descending order. There are three sorting techniques that can be used to sort the list in ascending or descending order. These are selection sort, bubble sort and insertion sort.

Selection Sort

The basic logic of selection sort is to repeatedly select the smallest element in the unsorted part of the array and then swap it with the first element of the unsorted part of the list. For example, consider the following

array

10 5 19 6 80 4 15

Step 1: The smallest element of the array, i.e. 4 is selected. This smallest element is then swapped with the first element of the array. So now the array becomes

4 5 19 6 80 10 15

sorted unsorted

Step 2: Now again the smallest element is picked up from the unsorted part of the array. The smallest element in the unsorted part is 5. Since 5 is already at the right position, so the array does not reflect any change.

4 5 19 6 80 10 15

Sorted Unsorted

Step 3: The next shortest element from the unsorted part is 6 which is swapped with 19. The array now becomes

4 5 6 19 80 10 15

Sorted Unsorted

Step 4: Next 10 is swapped with 19

4 5 6 10 19 80 15

Sorted Unsorted

Step 5: Now 15 is swapped with 19

4 5 6 10 15 80 19

Sorted Unsorted

Step 6: 19 is swapped with 80

4 5 6 10 15 19 80

Sorted

The array is finally sorted in ascending order.

The code for a selection sort involves two nested loops. The outside loop tracks the current position that the code wants to swap the smallest value with. The inside loop starts at the current location and scans the rest of the list in search of the smallest value. When it finds the smallest value, swapping of elements takes place.

Algorithm

1. small = DATA_LIST[0] #initialize small with the first element
2. for i=0 to U do
   {
3. small=DATA_LIST[i]
   position=i
4. for j=i to U do
   {
5. if DATA_LIST[j]<small then
6. { small = DATA_LIST[j]
7. position=j
   }
   j=j+1
   }
8. temp=DATA_LIST[i]
9. DATA_LIST[i]=small
10. DATA_LIST[position]=temp
    }
11. END

Selection Sort Program

```
def selection_sort(DATA_LIST):
    for i in range(0, len (DATA_LIST)):
    min = i
    for j in range(i + 1, len(DATA_LIST)):
        if DATA_LIST[j] < DATA_LIST[min]:
            min = j
    temp= DATA_LIST[min];
```

```
DATA_LIST[min] = DATA_LIST[i]
DATA_LIST[i]=temp            # swapping
```

Bubble Sort

In case of bubble sort algorithm, comparison starts from the beginning of the list. Every adjacent pair is compared and swapped if they are not in the right order (the next one is smaller than the former one). So, the heaviest element settles at the end of the list. After each iteration of the loop, one less element (the last one) is needed to be compared until there are no more elements left to be compared. Say for example the following array is to be sorted in ascending order

90 11 46 110 68 51 80

Step 1: Compare the first two elements, i.e. 90 and 11. Since they are not in order, so both these elements are swapped with each other. The array now becomes

11 90 46 110 68 51 80

Step 2: Compare second and third element, i.e. 90 and 46. Since they are not in order, so they are swapped. The array now becomes

11 46 90 110 68 51 80

Step 3: Compare 90 and 110. Since they are in order, so no change in the array.

Step 4: Compare 110 and 68. Since 68 is less than 110, so both are swapped

11 46 90 68 110 51 80

Step 5: Compare 110 and 51 . They need to be swapped.

11 46 90 68 51 110 80

Step 6: 110 and 80 are swapped since they are not in order.

11 46 90 68 51 80 110

After these six steps, the largest element is settled at its proper place i.e. at the end. So the first six elements form the unsorted part while the last element forms the sorted part. The above steps have to be repeated again to get the following array:

11 46 68 51 80 90 110

Unsorted sorted

Repeating the steps again, the array becomes

11 46 51 68 80 90 110

Unsorted sorted

The steps will be repeated again till all the elements are sorted in ascending order. The final sorted array is

11 46 51 68 80 90 110

Algorithm

```
1. for i=L to U
2. {    for j=L to ((U-1)-i)   #the unsorted array reduces with every iteration
3.      {    if(DATA_LIST[j]>DATA_LIST[j+1] then
4.           {    temp=DATA_LIST[j]
5.                DATA_LIST[j]=DATA_LIST[j+1]
6.                DATA_LIST[j+1]=temp
           }
      }
   }
7. END
```

Bubble Sort Program

```
#Bubble Sort
def bubble_Sort(DATA_LIST):
i = 0
j = 0
for i in range(len(DATA_LIST)):
    for j in range(len(DATA_LIST) - i):
        if DATA_LIST[j] < DATA_LIST[j-1]:
            temp = DATA_LIST[j-1]
            DATA_LIST[j-1] = DATA_LIST[j]
            DATA_LIST[j] = temp
    print DATA_LIST
print DATA_LIST
```

Insertion Sort

As far as the functioning of the outer loop is considered, insertion sort is similar to selection sort. The difference is that the insertion sort does not select the smallest element and put it into place; rather it selects the next element to the right of what was already sorted. Then it slides up each larger element until it gets to the correct location to insert into the sorted part of the array.

Consider the following unsorted array:

70 49 31 6 65 15 51

Unsorted array

Step 1: First element, i.e. 70 is compared with second element i.e.49 and swapped

49 70 31 6 65 15 51

Step 2: Third element, 31 compared with 70 and swapped.

49 31 70 6 65 15 51

Step 3: Further 31 is also swapped with 49 because 31 is less than 49

31 49 70 6 65 15 51

Step 4: Next element, 6 is swapped with 70, then 49, then 31

6 31 49 70 65 15 51

Step 4: 65 swapped with 70

6 31 49 65 70 15 51

Step 5: 15 swapped with 70, then 65, then 49, and then 31 to be inserted in the appropriate position

6 15 31 49 65 70 51

Step 6: 51 is inserted at proper position after being swapped by 70 and then by 65

6 15 31 49 51 65 70

As is visible from the above example, the insertion sort also breaks the list into two sections, the "sorted" half and the "unsorted" half. In each round of the outside loop, the algorithm will grab the next unsorted element and insert it into the sorted part of the list.

In the code below, the variable K marks the boundary between the sorted and unsorted portions of the list. The algorithim scans to the left of K using the variable ptr. Note that in the insertion short, ptr goes down to the left, rather than up to the right. Each cell location that is larger than keyvalue, temp gets moved up (to the right) one location. When the loop finds a location smaller than temp, it stops and puts temp to the left of it.

The algorithm and program for insertion sort is given below:

Algorithm

1. ctr=0
2. repeat steps 3 through 8 for K=1 to size-1
   {
3. temp=DATA_LIST[k]
4. ptr=K-1
5. repeat steps 6 to 7 while temp<DATA_LIST[ptr]
   {
6. DATA_LIST[ptr+1]=DATA_LIST[ptr]
7. ptr=ptr-1
   }
8. DATA_LIST[ptr+1]=temp
   }
9. END

Program

```
def insertion_sort(DATA_LIST):
    for K in range (1, len(DATA_LIST)):
        temp=DATA_LIST[K]
        ptr=K-1
        while(ptr>=0)AND DATA_LIST[ptr]>temp:
            DATA_LIST[ptr+1]=DATA_LIST[ptr]
            ptr=ptr-1
        DATA_LIST[ptr+1]=temp
```

LET'S REVISE

- Data structure: A group of data which can be processed as a single unit.
- There are two types of data structures - Linear and Non linear.
- Array: a set of contiguous data of similar data type.
- Python lists are actually arrays of variable length and have elements of different data types.
- Sequential allocation of memory: Elements stored in sequence of their declaration.
- Traversal: To move in a list in a sequential manner starting from first element to the last element.
- Insertion of a new element in a sorted list has to be inserted at an appropriate position so that it falls in order with the other elements of the list.
- Searching algorithms - Linear Search and Binary Search.
- In linear search, the search element is compared with each element of the list, starting from the beginning of the list to the end of the list.
- Binary Search: This searching technique reduces the number of comparisons and hence saves on processing time.
- For binary search, the array should be sorted in either ascending or descending order.
- Sorting is to arrange the list in an ascending or descending order.
- Sorting techniques - Selection, Bubble, Insertion
- Selection Sort: To repeatedly select the smallest element in the unsorted part of the array and then swap it with the first element of the unsorted part of the list.
- In case of bubble sort algorithm, every adjacent pair is compared and swapped if they are not in the right order
- Insertion Sort- Selects the next element to the right of what was already sorted, slides up each larger element until it gets to the correct location

EXERCISE

1. Define a data structure.
2. Name the two types of data structures and give one point of difference between them.
3. Give one point of difference between an array and a list in Python.
4. How are lists implemented in memory?
5. What is sequential allocation of memory? Why do we say that lists are stored sequentially?
6. How is memory allocated to a list in Python?
7. Write a function that takes a list that is sorted in ascending order and a number as arguments. The function should do the following:

a. Insert the number passed as argument in a sorted list.

b. Delete the number from the list.

8. How is linear search different from binary search?
9. Accept a list containing integers randomly. Accept any number and display the position at which the number is found is the list.
10. Write a function that takes a sorted list and a number as an argument. Search for the number in the sorted list using binary search.
11. In the following list containing integers, sort the list using Insertion sort algorithm. Also show the status of the list after each iteration.

    15 -5 20 -10 10

12. Consider the following unsorted list

    Neena Meeta Geeta Reeta Seeta

Sort the list using selection sort algorithm. Show the status of the list after every iteration.

13. Consider the following unsorted list

    90 78 20 46 54 1

Write the list after:

a. 3rd iteration of selection sort

b. 4th iteration of bubble ort

c. 5th iteration of insertion sort

14. Consider the following unsorted list:

10 5 55 13 3 49 36

Write the position of elements in the list after:

a. 5th iteration of bubble sort

b. 7th iteration of insertion sort

c. 4th iteration of selection sort

15. Sort a list containing names of students in ascending order using selection sort.

16. A list contains roll_no, name and marks of the student. Sort the list in descending order of marks using Selection Sort algorithm.

17. A list contains Item_code, Item_name and price. Sort the list :

a. In ascending order of price using Bubble sort.

b. In descending order of qty using Insertion sort.

18. Accept a list containing numbers. Sort the list using any sorting technique. Thereafter accept a number and display the position where that number is found. Also display suitable message, if the number is not found in the list.

Chpater-2: Stacks and Queues in List

Learning Objectives:

At the end of this chapter the students will be able to :

- Understand a stack and a queue
- Perform Insertion and Deletion operations on stacks and queues
- Learn Infix and Postfix expressions
- Convert an infix to postfix
- Evaluation of Postfix Expression

In the previous chapter you studied about manipulation of lists in Python. In this chapter you further study about stacks and queues and their implementation in Python using lists.

Stack

A stack is a data structure whose elements are accessed according to the Last-In First-Out (LIFO) principle. This is because in a stack, insertion and deletion of elements can only take place at one end, called top of the stack. Consider the following examples of stacks:

1. Ten glass plates placed one above another. (The plate that is kept last has to be taken out first)
2. The tennis balls in a container. (You cannot remove more than one ball at a time)
4. A pile of books
6. A stack of coins

http://us.123rf.com

In the above picture coins are kept one above the other and if any additional coin is to be added, it can be added only on the top. If we want to remove any coin from the stack, the coin on the top of the stack has to be removed first. That means, the coin that was kept last in the stack has to be taken out first.

The two operations performed on the stack are:

1. Push: Adding(inserting) new element on to the stack.
2. Pop: Removing (deleting) an element from the stack

Push operation

Adding new element to the stack list is called push operation. When the stack is empty, the value of top is

1. Basically, an empty stack is initialized with an invalid subscript.Whenever a Push operation is performed, the top is incremented by one and then the new value is inserted on the top of the list till the time the value of top is less than or equal to the size of the stack. Let us first have a look at the logic to program the Push operation for a stack through the following algorithm:

    1. Start
    2. Initialize top with -1.

    Step 3: Input the new element.

    Step 4: Increment top by one.

    Step 5: stack[top]=new element

    Step 6: Print "Item Inserted"

    Step 7: Stop

Pop operation

Removing existing elements from the stack list is called pop operation. Here we have to check if the stack is empty by checking the value of top. If the value of top is -1, then the stack is empty and such a situation is called Underflow. Otherwise Pop operation can be performed in the stack. The top is decremented by one if an element is deleted from the list. The algorithm for pop operation is as follows:

Algorithm for Pop operation:

Step 1: Start

Step 2: If the value of top is -1 go to step 3 else go to step 4

Step 3: Print "Stack Empty" and go to step 7

Step 4: Deleted item = Stack[top]

Step 5: Decrement top by 1

Step 6: print "Item Deleted"

Step 7: Stop

Traversal in a stack

Traversal is moving through the elements of the stack. If you want to display all the elements of the stack, the algorithm will be as follows:

Step 1: Start
Step 2: Check the value of top. If top=-1 go to step 3 else go to step 4
Step 3: Print "Stack Empty" and go to step 7
Step 4: Print the top element of the stack.
Step 5: Decrement top by 1
Step 6: If top= -1 go to step 7 else go to step 4
Step 7: Stop

In Python, we already have pop() and append() functions for popping and adding elements on to the stack. Hence, there is no need to write the code to add and remove elements in the stack. Consider the following programs that perform the Push and Pop operation on the stack through append() and pop().

Program to implement stack (without classes)

```
s=[]
c="y"
while (c=="y"):
   print "1. PUSH"
   print "2. POP "
   print "3. Display"
   choice=input("Enter your choice:  ")
   if (choice==1):
      a=input("Enter any number      :")
      s.append(a)
   elif (choice==2):
      if (s==[]):
         print "Stack Empty"
      else:
         print "Deleted element is :   ",s.pop()
   elif (choice==3):
      l=len(s)
      for i in range(l-1,-1,-1):
          print s[i]
   else:
      print("Wrong Input")
   c=raw_input("Do you want to continue or not?    ")
```

Output:

```
  >>>
1. PUSH
2. POP
3. Display
Enter your choice: 2
Stack Empty
Do you want to continue or not?  y
1.PUSH
2.POP
3.Display
Enter your choice: 1
Enter any number    :100
Do you want to continue or not?  y
1.  PUSH
2. POP
3. Display
Enter your choice: 1
Enter any number    :200
Do you want to continue or not?  y
1. PUSH
2. POP
3. Display
Enter your choice: 1
Enter any number    :300
Do you want to continue or not?  y
1. PUSH
2. POP
3. Display
Enter your choice: 3
300
200
100
```

```
Do you want to continue or not?  y
1. PUSH
2. POP
3. Display
Enter your choice:  2
Deleted element is :   300
Do you want to continue or not?  y
1. PUSH
2. POP
3. Display
Enter your choice:  3
200
100
Do you want to continue or not?  n
>>>
```

The same program can also be implemented using classes as shown below:

Program to implement a stack(Using classes)

```
class stack:
   s=[]
   def push(self):
      a=input("Enter any number     :")
      stack.s.append(a)
   def display(self):
      l=len(stack.s)
      for i in range(l-1,-1,-1):
          print stack.s[i]
a=stack()
c="y"
while (c=="y"):
  print "1. PUSH"
  print "2. POP "
  print "3. Display"
  choice=input("Enter your choice:  ")
  if (choice==1):
```

```
    a.push()
  elif (choice==2):
    if (a.s==[]):
      print "Stack Empty"
    else:
      print "Deleted element is :  ",a.s.pop()
  elif (choice==3):
    a.display()
  else:
    print("Wrong Input")
  c=raw_input("Do you want to continue or not?    ")output:
```

Output:

```
>>>
1. PUSH
2. POP
3. Display
Enter your choice:  1
Enter any number      :100
Do you want to continue or not?    y
1. PUSH
2. POP
3. Display
Enter your choice:  1
Enter any number      :200
Do you want to continue or not?    y
1. PUSH
2. POP
3. Display
Enter your choice:  3
200
100
Do you want to continue or not?    y
1. PUSH
2. POP
```

```
3. Display
Enter your choice: 2
Deleted element is : 200
Do you want to continue or not? y
1. PUSH
2. POP
3. Display
Enter your choice: 2
Deleted element is : 100
Do you want to continue or not: y
1. PUSH
2. POP
3. Display
Enter your choice: 2
Stack Empty
Do you want to continue or not? n
>>>
```

Expression

You have already studied about expressions in class XI. An expression is a combination of variables, constants and operators. Expressions can be written in Infix, Postfix or Prefix notations.

Infix Expression: In this type of notation, the operator is placed between the operands.
For example: A+B, A*(B+C), X*Y/Z, etc

Postfix Expression: In this type of notation, the operator is placed after the operands.
For example: AB+, ABC+*,XYZ/*, etc.

Prefix Expression: In this type of notation, the operator is placed before the operands.
For example: +AB,*A+BC,*X/YZ, etc.

Conversion of an infix expression to postfix expression

The following algorithm shows the logic to convert an infix expression to an equivalent postfix expression:

Step 1: Start

Step 2: Add "("(left parenthesis) and ")" (right parenthesis) to the start and end of the expression(E).

Step 3: Push "("left parenthesis onto stack.

Step 4: Check all symbols from left to right and repeat step 5 for each symbol of 'E' until the stack becomes empty.

Step 5: If the symbol is:

i) an operand then add it to list.

ii) a left parenthesis "("then push it onto stack.

iii) an operator then:

a) Pop operator from stack and add to list which has the same or higher precedence than the incoming operator.

b) Otherwise add incoming operator to stack.

iv) A right parenthesis")" then:

a) Pop each operator from stack and add to list until a left parenthesis is encountered.

b) Remove the left parenthesis.

Step 6: Stop

Example 1:

A*(B+C)

Operator/Operand	Stack	Result (List)	Meaning
A	A		Operand moved to result list
*	*	A	Operator pushed to stack
(	*(	A	Open Parentheses pushed to stack
B	*(	AB	Operand moved to result list
+	*(+	AB	Operator pushed to stack
C	*(+	ABC	Operand moved to result list
)	*	ABC+	Close Parentheses encountered,pop operators up to open Parentheses and add it to the result list. Remove open parenthesis from the stack
		ABC+*	Expression empty - Final Result

The resultant postfix expression is A B C + *

Example 2:

A/B^C-D

Opeartor/Operand	Stack	Result (List)	Meaning
A	A		Operand moved to result list
/	/	A	Operator pushed to stack
B	/	AB	Operand moved to result list
^	/^	AB	Operator pushed to stack
C	/^	ABC	Operand moved to result list
-	-	ABC^/	As compared to - operator, ^and / has higher priority, so pop both operators and add them to the Result list
D	-	ABC^/D	Operand moved to result list
		ABC^/D-	Expression empty - Final Result

The resultant postfix expression is ABC^/D-

Evaluation of Postfix Expression

The algorithm to evaluate a postfix expression is as follows:

Step 1: Start

Step 2: Check all symbols from left to right and repeat steps 3 & 4 for each symbol of expression 'E' until all symbols are over.

i) If the symbol is an operand, push it onto stack.

ii) If the symbol is an operator then

a) Pop the top two operands from stack and apply an operator in between them.

b) Evaluate the expression and place the result back on stack.

Step 3: Set result equal to top element on the stack.

Step 4: Stop

Example 1:

6,5,2,*,10,4,+,+.-

Operator/Operand	Stack	Calculation	Meaning
6	6		Operand pushed tostack
5	65		Operand pushed to stack
2	652		Operand pushed tostack

*	6 10	5*2=10	Pop last two operands, perform the operation and push the result in stack
10	6 10	10	Operand pushed tostack
4	6 10	10 4	Operand pushed to stack
+	6 10 14	10+4=14	Pop last two operands, perform the operation and push the result on to stack
+	6 24	10+14=24	Pop last two operands, perform the operation and push the result on to stack.
-	-18	6-24=-18	Pop last two operands, perform the operation and push the result on to stack.

The answer is - 18

Example 2:

True False AND True True NOT OR AND

Operator /Operand	**Stack**	**Calculation**	**Meaning**
True	True	Operand pushed to stack	
False	True False		Operand pushed to stack
AND	False	TRUE AND FALSE = FALSE	Pop last two operands, perform the operation and push the result on to stack.
True	False True		Operand pushed to stack
True	False True True	Operand pushed to stack	
NOT	False True False	NOT TRUE=FALSE	Pop last two operands, perform the operation and push the result on to stack.
OR	FALSE TRUE	TRUE OR FALSE= TRUE	Pop last two operands, perform the operation and push the result on to stack.
AND	FALSE	FALSE AND TRUE = FALSE	Pop last two operands, perform the operation and push the result on to stack.

The answer is **False**

Queue

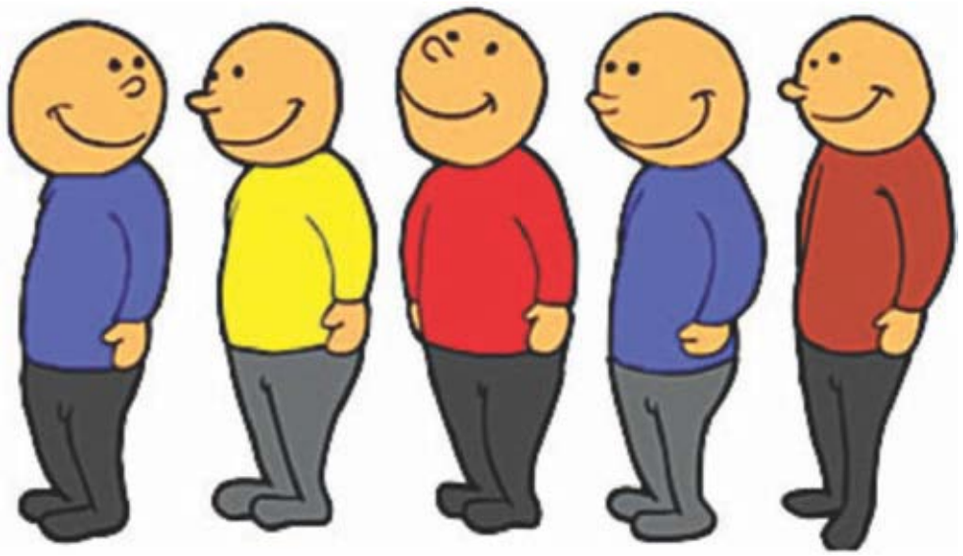
Another most common data structure found in computer algorithm(s) is queue. We are already familiar with it, as we run into enough of them in our day to day life. We queue up__

at the bank

at fee counter

at shopping centre etc.

A queue is a container of elements, which are inserted and removed according to the first-in first-out (FIFO) principle.



http://3.bp.blogspot.com/-

In a queue, persons who stand in the queue will carry out their work one by one. That means those who stands first in the queue will be allowed to carry out his work first and the person who stands at the second position will be allowed to carry out his work second only. At the same time those who come late will be joining the queue at the end. In simple terms it is called 'first come first out'.

Technically speaking a queue is a linear list, to keep an ordered collection of elements / objects. The principle operations, which can be performed on it are

- Addition of elements &
- Removal of elements.

Addition of element is known as INSERT operation, also known as enqueu-ing. It is done using rear terminal position, i.e. tail end. Removal of element is known as DELETE operation also know as dequeue-

ing. It is done using front terminal position, i.e. head of the list. As the two operations in the queue are performed from different ends, we need to maintain both the access points. These access points are known as FRONT, REAR. FRONT is first element of the queue and REAR is last element. As queue is FIFO implementation, FRONT is used for delete operation and REAR is used for insert operation.

Let's find out some applications of queue in computers:

- In a single processor multi tasking computer, job(s) waiting to be processed form a queue. Same happens when we share a printer with many computers.
- Compiling a HLL code
- Using down load manager, for multiple files also uses queue for ordering the files.
- In multiuser OS - job scheduling is done through queue.

Queue operations

Various operations, which can be performed on a queue are:

Create a queue having a data structure to store linear list with ordering of elements

Insert an element will happen using REAR, REAR will be incremented to hold the new value in queue.

Delete an element will happen using FRONT and FRONT will also be incremented to be able to access next element

Let's understand this with the help of an example:

1. We will use list data type to implement the queue.

2. As initially queue is empty, front and rear should not be able to access any element. The situation can be represented by assigning -1 to both REAR and FRONT as initial value.

F(front) = -1, **R**(rear) = -1

Once the queue is created, we will perform various operations on it. Following is the list of operations with its affect on queue:

INSERT(5)

F = F + 1

R = R + 1 (as this is the first element of the queue)

F R

INSERT(15)

R = R + 1

DELETE()

F = F + 1

INSERT(25)

R = R + 1

INSERT(35)

R = R + 1

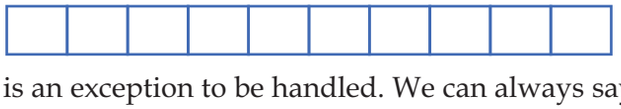
DELETE()

F = F + 1

DELETE()

F + F + 1

DELETE()



As the queue is empty, this is an exception to be handled. We can always say that deletion is attempted from an empty queue, hence not possible. The situation is known as **underflow** situation. Similarly when we work with fixed size **list,** insertion in a full list results into **overflow** situation. In python as we don't

have fixed size list, so don't need to bother about overflow situation.

Following are the formal steps for INSERT and DELETE operations

Algorithm for insertion:

Step 1: Start

Step 2: Check FRONT and REAR value, if both the values are -1, then
FRONT and REAR are incremented by 1
other wise
Rear is incremented by one.

Step 3: Add new element at Rear. (i.e.) queue[Rear]=new element.

Step 4: Stop

Algorithm for deletion:

Step 1: Start

Step 2: Check for underflow situation by checking value of Front = -1
If it is display appropriate message and stop
Otherwise

Step 3: Deleted item = queue [Front]

Step 4: If Front = Rear then Front =Rear = -1
Otherwise
Front is incremented by one

Step 5: Print "Item Deleted"

Step 6: Stop

Althogh principle operations in queue are Insert and Delete, but as a learner, we need to know the contents of queue at any point of time. To handle such requirement we will add traversal operation in our program. Following is the algorithm for same.

Algorithm for Display (Traversal in stack):

1. Start
2. Store front value in I
3. Check I position value, if I value is -1 go to step 4 else go to step 5
4. Print "Queue Empty" and go to step 8
5. Print queue[I]
6. I is incremented by 1
7. Check I position value, if I value is equal to rear+1 go to step 8 else go to step 5

Step 8: Stop

Note: In Python already we have **del()** and **append()** functions for deletion of elements at the front and addition of elements at the rear. Hence, no need of writing special function for add and remove elements in the queue. Likewise, 'Front and Rear positions' are also not required in the Python programming while implementing queue.

Example:

Write a program to implement Queue using list.

Code: (without using class)

```
a=[]
c='y'
while c=='y':
   print "1. INSERT"
   print "2. DELETE "
   print "3. Display"
   choice=input("enter your choice    ")
   if (choice==1):
         b=input("enter new number   ")
         a.append(b)
   elif (choice==2):
      if (a==[]):
         print("Queue Empty")
      else:
         print "deleted element is:",a[0]
         del a[0]
   elif (choice==3):
      l=len(a)
      for i in range(0,l):
          print a[i]
   else:
      print("wrong input")
   c=raw_input("do you want to continue or not   ")
```

Program: (Using class)

```
class queue:
```

```
    q=[]
    def insertion(self):
        a=input("enter any number:     ")
        queue.q.append(a)
    def deletion(self):
        if (queue.q==[]):
            print "Queue empty"
        else:
            print "deleted element is: ",queue.q[0]
            del queue.q[0]
    def display(self):
        l=len(queue.q)
        for i in range(0,l):
            print queue.q[i]
a=queue()
c="y"
while (c=="y"):
    print "1. INSERTION"
    print "2. DELETION "
    print "3. DISPLAY"
    choice=input("enter your choice:    ")
    if (choice==1):
        a.insertion()
    elif (choice==2):
        a.deletion()
    elif (choice==3):
        a.display()
    else:
        print("wrong input")
    c=raw_input("do you want to continue or not :")
```

LET'S REVISE

- **LIFO:** Last-In First-Out
- **FIFO:** First-In First-Out
- **Stack:** A stack is a container of elements that are inserted and removed according to the last-in first-out (LIFO) law.
- **Queue:** A queue is a container of elements, which are inserted and removed according to the first-in first-out (FIFO) law.
- **Infix Expression:** Operator is in between the operand.
- **Postfix Expression:** Operators are written after the operand.
- **Prefix Expression:** Operators are written before operand.

EXERCISE

1. Expand the following:
   (i) LIFO (ii) FIFO
2. What is stack?
3. What is Queue?
4. What are all operations possible in data structure?
5. Give one example of infix expression.
6. Give one example of postfix expression.
7. Give one example of prefix expression.
8. Convert (A+B)*C in to postfix form.
9. Evaluate using stack 10, 3,*,30,2,*,-
10. Converting following Infix expression to Postfix notation:
    a) (A+B)*C+D/E-F
    b) P+Q*(R-S)/T
    c) (True And False) || (False And True)
11. Evaluation the following Postfix Expression:
    a) 20,8,4,/,2,3,+,*,-
    b) 15,3,2,+,/,7,+,2,*
    c) False, Not, True, And, True, False, Or, And
    d) True, False, Not, And, False, True, Or, And
12. Write the push operation of stack containing names using class.
13. Write the pop operation of stack containing numbers using class.
14. Write the insertion operation of queue containing character using class.
15. Write the deletion operation of queue containing numbers using class.
16. Write any two example of stack operation.
17. Write any two example of pop operation.
18. Write an algorithm to evaluate postfix expression.
19. Write an algorithm to convert infix to postfix.
20. Write an algorithm to implement push operation.
21. Write an algorithm to implement pop operation.
22. Write an algorithm to implement insertion operation of queue.
23. Write an algorithm to implement deletion operation of queue.
24. Write a function to push any student's information to stack.
25. Write a function to add any customer's information to queue.

Chapter-3: Data File Handling

Learning Objective

After going through the chapter, student will be able to:

- Understand the importance of data file for permanent storage of data
- Understand how standard Input/Output function work
- Distinguish between text and binary file
- Open and close a file (text and binary)
- Read and write data in file
- Write programs that manipulate data file(s)

Programs which we have done so far, are the ones which run, produce some output and end. Their data disappears as soon as they stop running. Next time when you use them, you again provide the data and then check the output. This happens because the data entered is stored in primary memory, which is temporary in nature. What if the data with which, we are working or producing as output is required for later use? Result processing done in Term Exam is again required for Annual Progress Report. Here if data is stored permanently, its processing would be faster. This can be done, if we are able to store data in secondary storage media i.e. Hard Disk, which we know is permanent storage media. Data is stored using file(s) permanently on secondary storage media.You have already used the files to store your data permanently - when you were storing data in Word processing applications, Spreadsheets, Presentation applications, etc. All of them created data files and stored your data, so that you may use the same later on. Apart from this you were permanently storing your python scripts (as .py extension) also.

A file (i.e. data file) is a named place on the disk where a sequence of related data is stored. In python files are simply stream of data, so the structure of data is not stored in the file, along with data. Basic operations performed on a data file are:

- Naming a file
- Opening a file
- Reading data from the file
- Writing data in the file
- Closing a file

Using these basic operations, we can process file in many ways, such as

Creating a file

Traversing a file for displaying the data on screen

Appending data in file

Inserting data in file

Deleting data from file

Create a copy of file

Updating data in the file, etc.

Python allow us to create and manage two types of file

- Text
- Binary

A text file is usually considered as sequence of lines. Line is a sequence of characters (ASCII), stored on permanent storage media. Although default character coding in python is ASCII but using constant u with string, supports Unicode as well. As we talk of lines in text file, each line is terminated by a special character, known as End of Line (EOL). From strings we know that \n is newline character. So at the lowest level, text file will be collection of bytes. Text files are stored in human readable form and they can also be created using any text editor.

A binary file contains arbitrary binary data i.e. numbers stored in the file, can be used for numerical operation(s). So when we work on binary file, we have to interpret the raw bit pattern(s) read from the file into correct type of data in our program. It is perfectly possible to interpret a stream of bytes originally written as string, as numeric value. But we know that will be incorrect interpretation of data and we are not going to get desired output after the file processing activity. So in the case of binary file it is extremely important that we interpret the correct data type while reading the file. Python provides special module(s) for encoding and decoding of data for binary file.

To handle data files in python, we need to have a file object. Object can be created by using open() function or file() function. To work on file, first thing we do is open it. This is done by using built in function open(). Using this function a file object is created which is then used for accessing various methods and functions available for file manipulation.

Syntax of open() function is

file_object = open(filename [, access_mode] [,buffering])

open() requires three arguments to work, first one (filename) is the name of the file on secondary storage media, which can be string constant or a variable. The name can include the description of path, in case, the file does not reside in the same folder / directory in which we are working. We will know more about this in later section of chapter. The second parameter (access_mode) describes how file will be used throughout the program. This is an optional parameter and the default access_mode is reading. The third parameter (buffering) is for specifying how much is read from the file in one read. The function will return an object of file type using which we will manipulate the file, in our program. When we work with file(s), a buffer (area in memory where data is temporarily stored before being written to file), is automatically

associated with file when we open the file. While writing the content in the file, first it goes to buffer and once the buffer is full, data is written to the file. Also when file is closed, any unsaved data is transferred to file. flush() function is used to force transfer of data from buffer to file.

File access modes:

r will open the text file for reading only and **rb** will do the same for binary format file. This is also the default mode. The file pointer is placed at the beginning for reading purpose, when we open a file in this mode.

w will open a text file for writing only and **wb** for binary format file. The file pointer is again placed at the beginning. A non existing file will be created using this mode. Remember if we open an already existing file (i.e. a file containing data) in this mode then the file will be overwritten as the file pointer will be at the beginning for writing in it.

a mode allow us to append data in the text file and ab in binary file. Appending is writing data at the end of the file. In this mode, file pointer is placed at the end in an existing file. It can also be used for creating a file, by opening a non existing file using this mode.

r+ will open a text file and **rb+** will open a binary file, for both reading and writing purpose. The file pointer is placed at the beginning of the file when it is opened using r+ / **rb+** mode.

w+ opens a file in text format and **wb+** in binary format, for both writing and reading. File pointer will be placed at the beginning for writing into it, so an existing file will be overwritten. A new file can also be created using this mode.

a+ opens a text file and ab+ opens a binary file, for both appending and reading. File pointer is placed at the end of the file, in an already existing file. Using this mode a non existing file may be created.

Example usage of open

file= open("Sample.txt","r+")

will open a file called Sample.txt for reading and writing purpose. Here the name (by which it exists on secondary storage media) of the file specified is constant. We can use a variable instead of a constant as name of the file. Sample file, if already exists, then it has to be in the same folder where we are working now, otherwise we have to specify the complete path. It is not mandatory to have file name with extension. In the example .txt extension is used for our convenience of identification. As it is easy to identify the file as text file. Similarly for binary file we will use . dat extension.

Other function, which can be used for creation of a file is file(). Its syntax and its usage is same as open().

Apart from using open() or file() function for creation of file, **with statement** can also be used for same purpose. Using **with** ensures that all the resources allocated to file objects gets deallocated automatically once we stop using the file. Its syntax is :

with open() as fileobject :

Example:

with open("Sample.txt","r+") as file :

file manipulation statements

Let's know about other method's and function's which can be used with file object.

fileobject. close() will be used to close the file object, once we have finished working on it. The method will free up all the system resources used by the file, this means that once file is closed, we will not be able to use the file object any more. Before closing the file any material which is not written in file, will be flushed off. So it is good practice to close the file once we have finished using it. In case, if we reassign the file object to some other file, then python will automatically close the file.

Methods for reading data from the file are:

readline() will return a line read, as a string from the file. First call to function will return first line, second call next line and so on. Remember file object keeps the track of from where reading / writing of data should happen. For readline() a line is terminated by \n (i.e. new line character). The new line character is also read from the file and post-fixed in the string. When end of file is reached, readline() will return an empty string.

It's syntax is

fileobject.readline()

Since the method returs a string it's usage will be

>>>x = file.readline()

or

>>>print file.readline()

For reading an entire file using readline(), we will have to loop over the file object. This actually is memory efficient, simple and fast way of reading the file. Let's see a simple example of it

>>>for line in file:

… print line

Same can be achieved using other ways of looping.

readlines()can be used to read the entire content of the file. You need to be careful while using it w.r.t. size of memory required before using the function. The method will return a list of strings, each separated by \n. An example of reading entire data of file in list is:

It's syntax is:

fileobject.readlines()

as it returns a list, which can then be used for manipulation.

read() can be used to read specific size string from file. This function also returns a string read from the file. At the end of the file, again an empty string will be returned.

Syntax of read() function is

fileobject.read([size])

Here size specifies the number of bytes to be read from the file. So the function may be used to read specific quantity of data from the file. If the value of size is not provided or a negative value is specified as size then entire file will be read. Again take care of memory size available before reading the entire content from the file.

Let's see the usage of various functions for reading data from file. Assuming we have a file **data.txt** containing hello world.\n this is my first file handling program.\n I am using python language

Example of readlines():

```
>>>lines = []
>>>lines = file.readlines()
```

If we print element of lines (which can be done by iterating the contents of lines) we will get:

hello world.

this is my first file handling program.

I am using python language.

Can you notice, there are two blank lines in between every string / sentence. Find out the reason for it.

Example of using read():

```
lines = []
content = file.read()          # since no size is given, entire file will be read
lines = content.splitlines()
print lines
```

will give you a list of strings:

['hello world.', 'this is my first file handling program.', 'I am using python language.']

For sending data in file, i.e. to create / write in the file, **write() and writelines()** methods can be used. write() method takes a string (as parameter) and writes it in the file. For storing data with end of line character, you will have to add \n character to end of the string. *Notice addition of \n in the end of every sentence while talking of data.txt.* As argument to the function has to be string, for storing numeric value, we have to convert it to string.

Its syntax is

fileobject.write(string)

Example

```
>>>f = open('test1.txt','w')
>>>f.write("hello world\n")
>>>f.close()
```

For numeric data value conversion to string is required.

Example

```
>>>x = 52
>>>file.write(str(x))
```

For writing a string at a time, we use write() method, it can't be used for writing a list, tuple etc. into a file. Sequence data type can be written using writelines() method in the file. It's not that, *we can't write a string using writelines() method.*

It's syntax is:

fileobject.writelines(seq)

So, whenever we have to write a sequence of string / data type, we will use writelines(), instead of write().

Example:

```
f = open('test2.txt','w')
str = 'hello world.\n this is my first file handling program.\n I am using python language"
f.writelines(str)
f.close()
```

let's consider an example of creation and reading of file in interactive mode

```
>>>file = open('test.txt','w')
>>>s = ['this is 1stline','this is 2nd line']
>>>file.writelines(s)
>>>file.close()
>>>file.open('test.txt') # default access mode is r
>>>print file.readline()
>>>file.close()
```

Will display following on screen

this is 1stline this is 2nd line

Let's walk through the code. First we open a file for creation purpose, that's why the access mode is **w**. In the next statement a list of 2 strings is created and written into file in 3rd statement. As we have a list of 2 strings, writelines() is used for the purpose of writing. After writing the data in file, file is closed.

In next set of statements, first one is to open the file for reading purpose. In next statement we are reading the line from file and displaying it on screen also. Last statement is closing the file.

Although, we have used **readline()** method to read from the file, which is suppose to return a line i.e. string at a time, but what we get is, both the strings. This is so, because writelines() does not add any EOL character to the end of string. You have to do it. So to resolve this problem, we can create s using following statement

s = ['this is 1st line\n', 'this is 2nd line\n']

Now using readline(), will result into a string at a time.

All reading and writing functions discussed till now, work sequentially in the file. To access the contents of file randomly **- seek** and **tell** methods are used.

tell() method returns an integer giving the current position of object in the file. The integer returned specifies the number of bytes from the beginning of the file till the current position of file object.

It's syntax is

fileobject.tell()

seek()method can be used to position the file object at particular place in the file. It's syntax is :

fileobject.seek(offset [, from_what])

here offset is used to calculate the position of fileobject in the file in bytes. Offset is added to from_what (reference point) to get the position. Following is the list of from_what values:

Value	reference point
0	beginning of the file
1	current position of file
2	end of file

default value of from_what is 0, i.e. beginning of the file.

Let's read the second word from the test1 file created earlier. First word is 5 alphabets, so we need to move to 5th byte. Offset of first byte starts from zero.

```
f = open('test1.txt','r+')
f.seek(5)
fdata = f.read(5)
print fdata
f.close()
```

will display **world** on screen.

Let's write a function to create and display a text file using one stream object.

```
def fileHandling():
   file = open("story.txt","w+")
   while True:
      line = raw_input("enter sentence :")
      file.write(line)
      choice = raw_input("want to enter more data in file  Y / N")
      if choice.upper() == 'N' : break
   file.seek(0)
   lines = file.readlines()
   file.close()
   for l in lines:
      print l
```

in this function after opening the file, while loop allow us to store as many strings as we want in the file. once that is done, using seek() method file object is taken back to first alphabet in the file. From where we read the complete data in list object.

We know that the methods provided in python for writing / reading a file works with string parameters. So when we want to work on binary file, conversion of data at the time of reading, as well as writing is required. Pickle module can be used to store any kind of object in file as it allows us to store python objects with their structure. So for storing data in binary format, we will use pickle module.

First we need to import the module. It provides two main methods for the purpose, dump and load. For creation of binary file we will

use pickle.dump() to write the object in file, which is opened in binary access mode.

Syntax of dump() method is:

dump(object, fileobject)

Example:

```
def fileOperation1():
      import pickle
      l = [1,2,3,4,5,6]
      file = open('list.dat', 'wb')     # b in access mode is for binary file
      pickle.dump(l,file)           # writing content to binary file
      file.close()
```

Example:

Example of writing a dictionary in binary file:

```
MD = {'a': 1, 'b': 2, 'c': 3}
file = open('myfile.dat', 'wb')
pickle.dump(MD,file)
file.close()
```

Once data is stored using dump(), it can then be used for reading. For reading data from file we will: use pickle.load() to read the object from pickle file.
Syntax of load() is :

object = load(fileobject)

Note : we need to call load for each time dump was called.

```
# read python dict back from the file
ifile = open('myfile.dat', 'rb')
MD1 = pickle.load(ifile)          # reading data from binary file
ifile.close()
print MD1
```

Results into following on screen:

```
{'a': 1, 'c': 3, 'b': 2}
```

To distinguish a data file from pickle file, we may use a different extension of file. **.pk / .pickle are** commonly used extension for same.

Example of storing multiple integer values in a binary file and then read and display it on screen:

```
def binfile():
    import pickle                                    # line 1
    file = open('data.dat','wb')                     # line 2
    while True:
        x = int(raw_input())                         # line 3
        pickle.dump(x,file)                          # line 4
        ans = raw_input('want to enter more data Y / N')
        if ans.upper()== 'N' : break
    file.close()                                     # line 5
    file = open('data.dat','rb')                     # line 6
```

```
try :                                   # line 7
   while True :                         # line 8
      y = pickle.load(file)             # line 9
      print y                           # line 10
except EOFError :                       # line 11
   pass
file.close()                            # line 12
```

Let's walk through the code

Line 1 is importing pickle module, required to work on binary file. Line 2 is opening a binary file(data.dat) for writing in it. Using a loop we read integer value and then put it in file using line no. 3 & 4. In line number 5 we are closing the file - data.dat, this will de allocate all the resources being used with file. In line number 6 we are again associating data.dat to file stream for reading from it. Line number 7 & 11 is used for detection of end of file condition. This helps us in reading the content of entire file. try & except allow us to handle errors in the program. You will learn about them in details in next chapter. For now we know that, reading at end of file will result into error. Which is used here to terminate the loop used for reading the data from the file. Line no. 8 allow us to make an infinite loop, as the same will be handled using end of file condition. In line no. 9 we are getting data from the binary file and storing same in y, which is printed on screen in next line. Remember one load will get data for one dump. Last statement will again close the file.

Files are always stored in current folder / directory by default. The os (Operating System) module of python provides various methods to work with file and folder / directories. For using these functions, we have to import os module in our program. Some of the useful methods, which can be used with files in os module are as follows:

1. getcwd() to know the name of current working directory
2. path.abspath(filename) will give us the complete path name of the data file.
3. path.isfile(filename) will check, whether the file exists or not.
4. remove(filename) will delete the file. Here filename has to be the complete path of file.
5. rename(filename1,filename2) will change the name of filename1 with filename2.

Once the file is opened, then using file object, we can derive various information about file. This is done using file attributes defined in os module. Some of the attributes defined in it are

1. file.closed returns True if file is closed
2. file.mode returns the mode, with which file was opened.
3. file.name returns name of the file associated with file object while opening the file.

We use file object(s) to work with data file, similarly input/output from standard I/O devices is also performed using standard I/O stream object. Since we use high level functions for performing input/output through keyboard and monitor such as - raw_input(), input() and print statement we were not required to explicitly use I/O stream object. But let's learn a bit about these streams also.

The standard streams available in python are

- Standard input stream
- Standard output stream and
- Standard error stream

These standard streams are nothing but file objects, which get automatically connected to your program's standard device(s), when you start python. In order to work with standard I/O stream, we need to import sys module. Methods which are available for I/O operations in it are

- read() for reading a byte at a time from keyboard
- write() for writing data on terminal i.e. monitor

Their usage is

```
import sys
print ' Enter your name :'
name = ''
while True:
    c = sys.stdin.read()
    if c == '\n' :
      break
    name = name + c
    sys.stdout.write( 'your name is ' + name)
```

same can be done using high level methods also

```
name = raw_input('Enter your name :')
print 'your name is ',name
```

As we are through with all basic operations of file handling, we can now learn the various processes which can be performed on file(s) using these operations.

Creating a file

Option 1 : An empty file can be created by using open() statement. The content in the file can be stored later on using append mode.

Option 2 : create a file and simultaneously store / write the content also.

Algorithm

1. Open a file for writing into it
2. Get data to be stored in the file (can be a string at a time or complete data)
3. Write it into the file (if we are working on a string at a time, then multiple writes will be required.)
4. Close the file

Code :

```
def fileCreation():
        ofile = open("story.txt","w+")
        choice = True
        while True:
            line = raw_input("enter sentence :")
            ofile.write(line)
            choice = raw_input("want to enter more data in file  Y / N")
            if choice == 'N' : break
        ofile.close()
```

At the run time following data was provided
this is my first file program
writing 2nd line to file
this is last line

Traversal for display

Algorithm

1. Open file for reading purpose
2. Read data from the file (file is sequentially accessed by any of the read methods)
3. Display read data on screen
4. Continue step 2 & 3 for entire file
5. Close the file.

Program Code:

```
def fileDisplaying1():
  for l in open("story.txt","r").readlines():
        print l
file.close()
```

The above code will display

this is my first file programwriting 2nd line to filethis is last line.

Remember we didn't add new line character with string, while writing them in file.

```
def fileDisplaying2():
    file = open("story.txt","r")
    l = file.readline()
    while l:
        print l
        l = file.readline()
file.close()
```

The above code will also display the same content, i.e.

this is my first file programwriting 2nd line to filethis is last line.

```
def fileDisplaying3():
    file = open("story.txt","r")
    l = "123"
    while l != '':
        l = file.read(10)
        print l
file.close()
```

The above code will give the following output

this is my
first fil
e programw
riting 2nd
line to f
ilethis is
last line

Till now we were creating file object to either read from the file or to write in the file. Let's create an object capable of both reading and writing :

```
def fileHandling():
    file = open("story.txt","w+")          # both reading & writing can be done
```

```
    choice = True
    while True:
      line = raw_input("enter sentence :")
      file.write(line)                    #  creation of file
      choice = raw_input("want to enter more data in file  Y / N")
      if choice == 'N' : break
    file.seek(0)                    # transferring file object to beginning of the file
    lines = file.readlines()
    file.close()
    for l in lines:
          print l
```

The following code performs same operation a binary file

```
def binfile():
    import pickle
    file = open('data.dat','wb')
    while True:
      x = int(raw_input())
      pickle.dump(x,file)
      ans = raw_input('want to enter more data Y / N')
      if ans.upper()== 'N' : break
    file.close()
    file = open('data.dat','rb')
    try :
      while True :
          y = pickle.load(file)
          print y
    except EOFError :
      pass
    file.close()
```

Creating a copy of file

Algorithm

1. Open the source file for reading from it.
2. Open a new file for writing purpose
3. Read data from the first file (can be string at a time or complete data of file.)
4. Write the data read in the new file.
5. Close both the files.

Program Code for creating a copy of existing file:

```
def fileCopy():
  ifile = open("story.txt","r")
  ofile = open("newstory.txt","w")
  l = file.readline()
while l:
  ofile.write(l)
  l = file.readline()
ifile.close()
ofile.close()
```

Similarly a binary file can also be copied. We don't need to use dump() & load()methods for this. As we just need to pass byte strings from one file to another.

Deleting content from the file

It can be handled in two ways

Option 1 (for small file, which can fit into memory i.e. a few Mb's)

Algorithm

1. Get the data value to be deleted. (Nature of data will be dependent on type of file)
2. Open the file for reading from it.
3. Read the complete file into a list
4. Delete the data from the list
5. Close the file
6. Open same file for writing into it
7. Write the modified list into file.
8. Close the file.

Program code for deleting second word from story.txt is:

```
def filedel():
    with open('story.txt','r') as file :
      l = file.readlines()
    file.close()
    print l
    del l[1]
    print l
    file.open('story.txt','w')
    file.writelines(l)
    file.close()
```

Similarly we can delete any content, using position of the data. If position is unknown then serach the desired data and delete the same from the list.

Option 2 (for large files, which will not fit into memory of computer. For this we will need two files)

Algorithm

1. Get the data value to be deleted
2. Open the file for reading purpose
3. Open another (temporary) file for writing into it
4. Read a string (data value) from the file
5. Write it into temporary file, if it was not to be deleted.
6. The process will be repeated for entire file (in case all the occurrence of data are to be deleted)
7. Close both the files.
8. Delete the original file
9. Rename the temporary file to original file name.

Program code for deletion of the line(s) having word (passed as argument) is :

```
import os
def fileDEL(word):
    file = open("test.txt","r")
    newfile = open("new.txt","w")
    while True:
      line = file.readline()
      if not line :
          break
```

```
    else :
    if word in line :
        pass
    else :
        print line
        newfile.write(line)
file.close()
newfile.close()
os.remove("test.txt")
os.rename("new.txt","test.txt")
```

Inserting data in a file

It can happen in two ways. Insertion at the end of file or insertion in the middle of the file.

Option 1 Insertion at the end. This is also known as appending a file.

Algorithm

1. open the file in append access mode.
2. Get the data value for to be inserted in file, from user.
3. Write the data in the file
4. Repeat set 2 & 3 if there is more data to be inserted (appended)
5. Close the file.

Program code for appending data in binary file

```
def binAppend():
    import pickle
    file = open('data.dat','ab')
    while True:
        y = int(raw_input())
        pickle.dump(y,file)
        ans = raw_input('want to enter more data Y / N')
        if ans.upper()== 'N' : break
file.close()
```

Option 2 Inserting in the middle of file

There is no way to insert data in the middle of file. This is a limitation because of OS. To handle such insertions re-writing of file is done.

Algorithm

1. Get the data value to to be inserted with its position.
2. Open the original file in reading mode.
3. Open another (temporary) file for writing in it.
4. Start reading original file sequentially, simultaneously writing it in the temporary file.
5. This is to be repeated till you reach the position of insertion of new data value.
6. Write the new value in temporary file.
7. Repeat step 4 for remaining data of original file.
8. Delete original file
9. Change the name of temporary file to original file.

Code for inserting data in the file with the help of another file

It will be similar to the code used for deletion of content using another file. Here instead of not writing the content add it in the file.

An alternative to this is, first read the complete data from file into a list. Modify the list and rewrite the modified list in the file.

Updating a file

File updation can be handled in many ways. Some of which are

Option 1 - Truncate write

Algorithm

1. Open the file for reading from it
2. Read the content of file in an object (variable) usually list
3. Close the file
4. Get the details of data to be modified
5. Update the content in the list
6. Re open the file for writing purpose (we know that now opening the file for writing will truncate the existing file)
7. Write the list back to the file.

Program Code for this will be similar to following:

```
with open("sample.txt","r") as file:
  content = file.read()
file.close()
```

```
content.process()
wit h open ("sample.txt","w") as file :
  file.writelines(content)
file.close()
```

Option 2 - Write replace

Algorithm

- Open the original file for reading
- Open temporary file for writing
- Read a line / record from the file
- If this was not to be modified copy it in temporary file otherwise copy the modified line / record in the temporary file.
- Repeat previous two steps for complete file.

This way of processing a file using python has been handled earlier.

Option 3 - In place updation

Algorithm

1. Open file for reading and writing purpose
2. Get the details of data value to be modified
3. Using linear search, reach to the record / data to be modified
4. Seek to the start of the record
5. Re write the data
6. Close the file.

Updating a text file in this manner is not safe, as any change you make to the file may overwrite the content you have not read yet. This should only be used when the text to be replaced is of same size. In place updation of a binary file is not possible. As this requires placing of fileobject to the beginning of the record, calculating size of data in dump file is not possible. So updating a data file using third option is not recommended in python.

Let's create a data file storing students record such as Admission number, Name, Class and Total marks. Data to be stored contains numeric data, hence will be stored in binary file. We will use dictionary data type to organize this information.

```
from pickle import load, dump
import os
import sys
```

```
def bfileCreate(fname):
    l = []
    sd = {1000:['anuj',12,450]}
    with open(fname,'wb') as ofile :
      while True :
          dump(sd,ofile)
          ans = raw_input("want to enter more data Y / N")
          if ans.upper() == 'N' : break
          x = int(raw_input("enter admission number of student"))
          l = input("enter name class and marks of student enclosed in []")
          sd[x] = l
    ofile.close()

    def bfileDisplay(fname):
      if not os.path.isfile(fname) :
          print "file does not exist"
      else:
          ifile = open(fname,'rb')
          try :
              while True:
                  sd = {}
                  sd = load(ifile)
                  print sd
          except EOFError:
                  pass
    ifile.close()
```

Use the code to store records of your class mates. Once the file is created, use bfileDisplay() to see the result. Do you find some problem in the content displayed? Find and resolve the problem?

LET'S REVISE

Files are used to store huge collection of data permanently. The stored data can later be used by performing various file operations like opening, reading, writing etc.

Access modes specify the type of operations to be performed with the opened file.

read(), readline() and readlines() methods are available for reading data from the file.

write() and writelines() are used for writing data in the file.

There are two functions which allow us to access a file in a non-sequential or random mode. They are seek() and tell()

Serialization is the process of converting a data structure / object that can be stored in non string format and can be resurrected later. Serialization can also be called as deflating the data and resurrecting as inflating it.

Pickle module is used in serialization of data. This allow us to store data in binary form in the file. Dump and load functions are used to write data and read data from file.

os module provide us various functions and attributes to work on files.

EXERCISE

1.
```
file = open('textfile.txt','w')
word = ''
while word.upper() != 'END':
    word = raw_input('Enter a word use END to quit')
    file.write(word + '\n')
file.close()
```

The above program is to create a file storing a list of words. What is the name of file on hard disk containing list of words?

2. Human readable form of file is called ----------.
3. Write a try …. except statement that attempts to open a file for reading and catches the exception thrown when the file does not exist.
4. Compare & contrast read(), readline() and readlines().
5. How is write() different from writelines()?
6. In how many ways can end of file be detected?
7. How many file modes can be used with the open() function to open a file? State the function of each mode.
8. What does the seekg() and seekp() functions do?
9. Explain the use of output functions write() and writeline() with an example each.
10. Write a function that writes a structure to disk and then reads it back and display on screen.
11. Using the file in mentioned in question no. 1, write a function to read the file and display numbered list of words.
12. In the code (given in question no. 1) the word END used to indicate end of word list is also stored in the file. Modify the code so that end is not stored in the file.
13. Write a function that takes three argument 1st input file, 2nd output file and 3rd transformation function. First argument is the file opened for reading, second argument is the file opened for writing and third argument is a function, which takes single string and performs a transformation of your choice and returns the transformed string. The function should reed each line in the input file, pass the line through transformation function and then write transformed line to output file. A transformation can be - capitalize first alphabet of every word.

14. Write a function to create a text file containing following data

    Neither apple nor pine are in pineapple. Boxing rings are square.

    Writers write, but fingers don't fing. Overlook and oversee are opposites. A house can burn up as it burns down. An alarm goes off by going on.

    i) Read back the entire file content using read() or readlines () and display on screen.
    ii) Append more text of your choice in the file and display the content of file with line numbers prefixed to line.
    iii) Display last line of file.
    iv) Display first line from 10th character onwards
    v) Read and display a line from the file. Ask user to provide the line number to be read.

15. Create a dictionary having decimal equivalent of roman numerals. Store it in a binary file. Write a function to convert roman number to decimal equivalent using the binary file data.

16. Write a program to delete the content provided by user from the binary file. The file is very large and can't fit in computers memory.

17. Write a function to insert a sentence in a text file, assuming that text file is very big and can't fit in computer's memory.

18. Write a program to read a file 'Story.txt' and create another file, storing an index of Story.txt telling which line of the file each word appears in. If word appears more than once, then index should show all the line numbers containing the word.

Hint : Dictionary with key as word(s) can be used to solve this.

19. Write a function called replace file(), that takes pattern string, replacement string and two file names as argument. The function should read the first file and write the content into second file (creating it, if necessary). If the pattern string appear anywhere in first file, it should be replaced by replacement string in second file.

20. Write a program to accept a filename from the user and display all the lines from the file which contain python comment character '#'.

21. Reading a file line by line from beginning is a common task, what if you want to read a file backward. This happens when you need to read log files. Write a program to read and display content of file from end to beginning.

22. Create a class item to store information of different items, existing in a shop. At least following is to be stored w.r.t. each item code, name, price, qty. Write a program to accept the data from user and store it permanently in the file. Also provide user with facility of searching and updating the data in file based on code of item.

Chapter-4: Exception Handling & Generator Functions

Learning Objective

At the end of this chapter the students will be able to understand:

- How does Python deal with errors?
- What an exception is and its terminology.
- Why we use them?
- What are the different types of exception?
- Generating exceptions and handling multiple exceptions
- Distinguish between iterators and generators
- Create generator functions

When we plan our code/program, we always work for situations that are normally expected, and our program works very well in those situations. But, we all understand that programs have to deal with errors. Here errors are not syntax errors instead they are the unexpected condition(s) that are not part of normal operations planned during coding. Partial list of such kinds of errors are:

- Out of Memory
- Invalid filename
- Attempting to write into read only file
- Getting an incorrect input from user
- Division by zero
- Accessing an out of bound list element
- Trying to read beyond end of file
- Sending illegal arguments to a method

If any of such situations is encountered, a good program will either have the code check for them and perform some suitable action to remedy them, or at least stop processing in a well defined way after giving appropriate message(s). So what we are saying is if an error happened, there must be code written in program, to recover from the error. In case if it is not possible to handle the error then it must be reported in user friendly way to the user.

Errors are exceptional, unusual and unexpected situations and they are never part of the normal flow of a program. We need a process to identify and handle them to write a good program. Exceptions handling is the process of responding in such situations. Most of the modern programming languages provide support with handling exceptions. They offer a dedicated exception handling mechanism, which simplifies the way in which an exception situation is reported and handled.

Before moving ahead with exception handling, let's understand, some of the terms associated with it - when an exception occurs in the program, we say that exception was raised or thrown. Next, we deal with

it and say it is handled or caught. And the code written to handle it is known as exception handler.

For handling exceptional situations python provides

1. **raise statement** to raise exception in program
2. **try..... except** statement for catching and handling the errors.

Raise statement allows the programmer to force a specified exception to occur. Once an exception is raised, it's up to caller function to either handle it using try/except statement or let it propagate further.

Syntax of raise is:

raise [exception name [, argument]]

A raise statement that does not include an exception name will simply re raise the current exception.

Here exception name is the type of error, which can be string, class or object and argument is the value. As argument is optional its default value **None** is used in case the argument is not provided.

Before moving further, let's talk of some of the python's predefined error types, which we can use with raise statement. This does not mean that raise can't be used to raise user defined errors.

S. No.	Error type	Description
1.	IOError	is raised when I/O operator fails. Eg. **File not found, disk full**
2.	EOFError	is raised when, one of the file method i.e. read(), readline() or readlines(), try to read beyond the file.
3.	ZeroDivisionError	is raised when, in division operation, denominator is zero
4.	ImportError	is raised when import statement fails to find the module definition or file name
5.	IndexError	is raised when in a sequence - index/subscript is out of range.
6.	NameError	is raised when a local or global name is not found
7.	IndentationError	is raised for incorrect indentation
8.	TypeError	is raised when we try to perform an operation on incorrect type of value.
9.	ValueError	is raised when a built in function/method receives an argument of correct type but with inappropriate value.

Example:

```
>>>l = [1,2,3]
>>>i = 5
>>> if i > len(l):
```

```
        raise IndexError
    else:
        print l[i]
```

As the value assigned to i is 5, the code will raise an error, during execution. The error will be index error as mentioned in raise statement.

Let's see an example of raise, with user defined error type:

```
>>>def menu(choice):
        if choice < 1:
            raise "invalid choice", choice
```

In order to handle the errors raised by the statement, we can use try except statement.

try.....except is used when we think that the code might fail. If this happens then we are able to catch it and handle same in proper fashion. Let's re-consider the menu() function defined above to handle the error also:

```
while True:
    try:
        x = int(raw_input("Enter a number"))
        break
    except ValueError:
        print " This was not a valid number. Enter a valid number"
```

This code will ask user to input a value until a valid integer value is entered.

Now let's see how try and except work?

Once while loop is entered execution of try clause begins, if no error is encountered, i.e. the value entered is integer, except clause will be skipped and execution of try finishes. If an exception occurs i.e. an integer value is not entered then rest of try clause is skipped and except cause is executed.

Following is the syntax of try...except statement

try:
    statements which might go wrong
    except *error type1*:
    statements to be executed, if error type1 happens
[**except** *error type2*:
    statements to be executed, if error type 2 happens
    .
    .
    .

else:

statements to be executed, if no exception occurs

finally:

statements to be executed]

remember error type can be user defined also.

You can see a single try statement can have multiple except statements. This is required to handle more than one type of errors in the piece of code. In multiple except clauses, a search for except is taken up. Once a match is found it is executed. You may not specify an error type in exception clause. If that is done, it is to catch all exceptions. Such exceptions should be listed as last clause in the try block.

The **else** clause written after all except statements, is executed, if code in try block does not raise any exception.

finally clause is always executed before leaving the try statement irrespective of occurrence of exception. It is also executed if during execution of try and except statement any of the clause is left via break, continue or return statement.

In try statement the block of code written in try is main action statement, except clause defines handlers for exceptions raised during execution of main statements. Else clause, if written, provides a handler to be run if no exception occurs, and finally is used to provide clean up action.

Example using user defined exception, which was created earlier in raise statement

```
>>>try:
        Statement(s)
    except "Invalid choice ":
        exception handling statements
    else:
        rest of the code
```

Example of except clause without error type.

```
    >>>try:
    x = y
    except:
        print " y not defined "
```

Example code having except, else and finally clause

```
def divide(x, y):
...    try:
...        result = x / y
...    except ZeroDivisionError:
...        print "division by zero!"
```

```
... else:
...     print "result is", result
... finally:
...     print "executing finally clause"
```

Let's apply exception handling in data file.

```
try:
    fh = open("testfile", "w")
    fh.write("This is my test file for exception handling!!")
    except IOError:
    print "Error: can\'t find file or read data"
else:
    print "Written content in the file successfully"
```

This will produce the following result, if you are not allowed to write in the file :

Error: can't find file or read data

Another example of error handling in data file:

```
lists = [ ]
infile = open('yourfilename.pickle', 'r')
while 1:
    try:
        lists.append(pickle.load(infile))
    except EOFError:
        break
infile.close()
```

This will allow us to read data from a file containing many lists, a list at a time.

Generator Functions

Iteration is the repetition of a process in computer program. This is typically done using looping constructs. These loops will repeat a process until certain number / case is reached. Recursive function is other way of implementing iteration in the program. Sequence data type, in python is iterable objects.

Here itratable objects have a specific protocol, which allow us to iterate (loop) over different type of objects. Such as - list, strings, dictionaries, files and others. The purpose of the protocol is to allow a user to process every element of container, without knowing the internal structure of container. So irrespective of whether container is list, string, tuple or file we can iterate in a similar fashion on all of them. This you have already done in class XI using for statement, for could be used to iterate a string, list, dictionary and tuple.

Iteration protocol specifies that, an iterator should have at least three operators:

1. Increment

2. De referencing
3. Non equality comparison

We are not going to cover their details in the chapter instead we will talk of generators, which allow us to write user defined iterator, without worrying about the iterator protocol.

Before moving further into generator functions let's walk through the execution of normal function. When a normal python function is called, in a program, the control from calling function is shifted to called function. Execution of called function starts from first line and continues until a return statement or exception or end of function is encountered. In all these situations, the control is returned back to caller function. That means any work done by the called function for providing the result is lost. To use it again we will again call the function and its execution will start from scratch.

Sometimes in programming we need to have functions which are able to save its work, so that instead of starting from scratch the function starts working from the point where it was left last. In other words, a function should be able to yield a series of values instead of just returning a single value. Here returning a value also implies returning of control.

Such functions are Generator functions. These functions can send back a value and later resume processing from the place, where they left off. This allows the function to produce a series of values - over time, rather than computing them all at once and returning a list of values.

Generator functions are not much **different** from normal functions, they also use **def** to define a function. The primary difference between generator and normal function is that generator will yield a value instead of returning a value. It is the **yield** statement which allows the generator function to suspend the processing and send a value, simultaneously retaining the existing state of generator to resume processing over time.

Let's take a simple example illustrating yield

```
def testGenerator():
    yield 1
    yield 2
    yield 3
```

For using the generator function, following will be done

```
>>> g = testGenerator()
>>> g.next()
1
>>>g.next()
2
>>>g.next()
3
```

Let's consider another example:

```
1.  def counter (n):
2.      i = 1
3.      while i< = n:
4.          yield i
5.          i+=1
6.  >>> x= counter (3)
7.  >>> x.next ( )
8.  1
9.  >>> x. next ( )
10. 2
11. >>> x.next ( )
12. 3
```

Let's walk through the code, except for yield statement remaining statements are just like normal function

1. Presence of yield statement (at line no.4) means that the function is not normal function but a generator
2. Calling the function, does not actually execute the function, but creates an instance of the function. As done in line no. 6. Here counter ( ) is actually creating an object x.
3. In line no, 7, the next ( ) method, executes the code written in counter up to first yield statement and then returns the value generated. In our case, it will be 1, which is displayed in line no. 8.
4. Calling next ( ) again in line no. 9, will resume the processing of function counter, from the place it was last left till the yield statement is encountered again. So in our example processing will resume from line no. 5, and will continue to line no. 3 & 4. At 4, there is yield which will again return the generated value back, after pausing the processing.

Since counter is called with argument value 3, so the process will be repeated three times.

Trace the following code, and explain what is happening and why?

```
def Count ( ) :
    n = 1
    while True:
        yield n
        n+ = 1
```

Let's take an example of producing squares of **n** numbers using iterator and generator.

Squares using generator

```
def Square (n):
    for i in range (n):
        yield i**2
```

The function can be invoked in following way

```
>>> for k in square (6):
        print k,
```

Let's do same thing using iterator, i.e., we will replace yield with return

```
def Square (n):
    for i in range (n):
        return i**2
```

Calling the function square

```
>>> for k in square (6):
        print k
```

results into Type Error, saying int object is not iterable. It's because here Square( ) will just return an integer object not a series of values.

Remember using generator you can iterate on generated data only once.

Let's use generator to produce Fibonacci series.

```
def Fibonacci (max):
    a, b = 0, 1
    while a <= max:
        yield a
        a, b = b, a + b
>>> for i in Fibonacci (100):
        print i,
0 1 1 2 3 5 8 13 21 34 55 89
```

The generator can also be used to return a list of values

```
>>> L = list (Fibonacci (100))
>>> print L
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

Advantages of using generator

- These functions are better w.r.t. memory utilization and code performance, as they allow function to avoid doing all work at a time.
- They provide a way to manually save the state between iterations. As the variables in function scope are saved and restored automatically.

LET'S REVISE

- An exception is a rarely occurring condition that requires deviation from the program's normal flow.
- We can raise and handle the errors in our program
- For raising errors statement used is raise[exception name [, argument]]
- For handling errors statement used is try …. except…… else finally.
- Iterable is an object capable of returning its member one at a time.
- An iterator is an object that provides sequential access to an underlying sequential data. The underlying sequence of data is not stored in memory, instead computed on demand.
- A generator is user defined iterator.
- A generator is a function that produces a sequence of results instead of simple value using yield statement

EXERCISE

1. What all can be possible output's of the following code

```
def myfunc(x=None):
    result = ""
    if x is None:
    result = "No argument given"
    elif x == 0:
    result = "Zero"
    elif 0 < x <= 3:
    result = "x is between 0 and 3"
else:
    result = "x is more than 3"
return result
```

2. We will try to read input from the user. Press ctrl-d and see what happens

```
>>> s = raw_input('Enter something --> ')
```

3. What will happen when following functions are executed?

```
def fib():
x,y = 1,1
while True:
yield x
x,y = y, x+y
def odd(seq):
for number in seq:
if number % 2:
yield number
def under Four Million(seq):
for number in seq:
```

```
    if number > 4000000:
        break
    yield number
        print sum(odd(underFourMillion(fib())))
```

4. Find out the situation(s) in which following code may crash

```
while loop == 1:
try:
    a = input('Enter a number to subtract from > ')
    b = input ('Enter the number to subtract > ')
    except NameError:
    print "\nYou cannot subtract a letter"
    continue
except  SyntaxError:
print  "\nPlease enter a number only."
continue
print a - b
try:
loop = input('Press 1 to try again > ')
except (NameError,SyntaxError):
loop = 0
```

5. Describe, what is the following module doing:

```
def test(start = 0):
c = start
while True:
value = yield c
if value != None:
c = value
    else:
        c += 1
```

6. When do we get type error?
7. List the situation(s) in which the code may result in IOError?
8. What is the purpose of yield statement?
9. How is yield different from return?
10. Write syntax of raise statement.
11. Write a function called oops that explicitly raises a Index Error exception when called. Then write another function that calls oops inside a try/except statement to catch the error. What happens if you change oops to raise Key Error instead of Index Error?
12. range() function in python does not include the stop value. Write a generator function equivalent to range() function that includes the stop value also. The function will take three arguments start, stop and step and will generate the desired list.
13. Write a function to find average of a list of numbers. Your function should be able to handle an empty list and also list containing string.
14. Write a function to generate cube's of numbers over time.
15. Write a program, to accept a date as day, month & year from user and raise appropriate error(s), if legal value(s) is not supplied. Display appropriate message till user inputs correct value(s).
16. Create a class Person to store personal information (of your choice) for a person. Ensure that while accepting the data incorrect entry is properly handled.