CHAPTER 10

LISTS AND TUPLES

A sequence is a datatype that represents a group of elements. The purpose of any sequence is to store and process a group of elements. In Python, strings, lists, tuples and dictionaries are very important sequence datatypes. All sequences allow some common operations like indexing and slicing. In this chapter, you will learn about lists, tuples and their operations in Python.

List

A list is similar to an array that consists of a group of elements or items. Just like an array, a list can store elements. But, there is one major difference between an array and a list. An array can store only one type of elements whereas a list can store different types of elements. Hence lists are more versatile and useful than an array. Perhaps lists are the most used datatype in Python programs.

In our daily life, we do not have elements of the same type. For example, we take marks of a student in 5 subjects:

```
50, 55, 62, 74, 66
```

These are all belonging to the same type, i.e. integer type. Hence, we can represent such elements as an array. But, we need other information about the student, like his roll number, name, gender along with his marks. So, the information looks like this:

```
10, Venu gopal, M, 50, 55, 62, 74, 66
```

Here, we have different types of data. Roll number (10) is an integer. Name ('Venu gopal') is a string. Gender ('M') is a character and the marks (50, 55, 62, 74, 66) are again integers. In daily life, generally we have this type of information that is to be stored and processed. This type of information cannot be stored in an array because an array can store only one type of elements. In this case, we need to go for list datatype. A list can

store different types of elements. To store the student's information discussed so far, we can create a list as:

```
student = [10, 'Venu gopal', 'M', 50, 55, 62, 74, 66]
```

Please observe that the elements of the 'student' list are stored in square braces []. We can create an empty list without any elements by simply writing empty square braces as:

```
e_lst = []   # this an empty list
```

Thus we can create a list by embedding the elements inside a pair of square braces []. The elements in the list should be separated by a comma (,). To view the elements of a list as a whole, we can simply pass the list name to the print() function as:

```
print(student)
```

The list appears as given below:

```
[10, 'Venu gopal', 'M', 50, 55, 62, 74, 66]
```

Indexing and slicing operations are commonly done on lists. Indexing represents accessing elements by their position numbers in the list. The position numbers start from 0 onwards and are written inside square braces as: student[0], student[1], etc... It means, student[0] represents 0th element, student[1] represents 1st element and so forth. For example, to print the student's name, we can write:

```
print(student[1])
```

The name of student appears as given below:

```
Venu gopal
```

Slicing represents extracting a piece of the list by mentioning starting and ending position numbers. The general format of slicing is: [start: stop: stepsize]. By default, 'start' will be 0, 'stop' will be the last element and 'stepsize' will be 1. For example, student[0:3:1] represents a piece of the list containing 0th to 2nd elements.

```
print(student[0:3:1])
```

The elements are given below:

```
[10, 'Venu gopal', 'M']
```

We can also write the above statement as:

```
print(student[:3:])
```

The same elements appears as shown following:

```
[10, 'Venu gopal', 'M']
```

Here, since we did not mention the starting element position, it will start at 0 and stepsize will be taken as 1. Suppose, we do not mention anything in slicing, then the total list will be extracted as:

```
print(student[::])
```

It displays the output as:

```
[10, 'Venu gopal', 'M', 50, 55, 62, 74, 66]
```

Apart from indexing and slicing, the 5 basic operations: finding length, concatenation, repetition, membership and iteration operations can be performed on lists and other sequences like strings, tuples or dictionaries.

In the following program, we are creating lists with different types of elements and also displaying the list elements.

Program

Program 1: A Python program to create lists with different types of elements.

```
# a general way to create lists
# create a list with integer numbers
num = [10, 20, 30, 40, 50]
print('Total list= ', num)  # display total list
print('First= %d, Last= %d' % (num[0], num[4]))  # display first and
     # last elements

# create a list with strings
names = ["Raju", "Vani", "Gopal", "Laxmi"]
print('Total list= ', names)  # display entire list
print('First= %s, Last= %s' % (names[0], names[3]))  # display first
     # and last elements

# create a list with different elements
x = [10, 20, 10.5, 2.55, "Ganesh", 'Vishnu']
print('Total list= ', x)  # display entire list
print('First= %d, Last= %s' % (x[0], x[5]))  # display first and last
     # elements
```

Output:

```
C:\>python lists.py
Total list=  [10, 20, 30, 40, 50]
First= 10, Last= 50
Total list=  ['Raju', 'Vani', 'Gopal', 'Laxmi']
First= Raju, Last= Laxmi
Total list=  [10, 20, 10.5, 2.55, 'Ganesh', 'Vishnu']
First= 10, Last= Vishnu
```

Creating Lists using range() Function

We can use range() function to generate a sequence of integers which can be stored in a list. The format of the range() function is:

```
range(start, stop, stepsize)
```

If we do not mention the 'start', it is assumed to be 0 and the 'stepsize' is taken as 1. The range of numbers stops one element prior to 'stop'. For example,

```
range(0, 10, 1)
```

This will generate numbers from 0 to 9, as: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]. Consider another example:

```
range(4,9,2)
```

The preceding statement will generate numbers from 4th to 8th in steps of 2, i.e. [4, 6, 8].

In fact, the range() function does not return list of numbers. It returns only range class object that stores the data about 'start', 'stop' and 'stepsize'. For example, if we write:

```
print(range(4,9,2))
```

The preceding statement will display:

```
range(4, 9, 2)   # this is the object given by range()
```

This range object should be used in for loop to get the range of numbers desired by the programmer. For example:

```
for i in range(4,9,2):
    print(i)
```

The preceding statement will display 4, 6, 8. Hence, we say range object is 'iterable', that is, suitable as a target for functions and loops that expect something from which they can obtain successive items. For example, range object can be used in for loops to display the numbers, or with list() function to create a list. In the following example, the range() function is used in the list() function to create a list:

```
lst = list(range(4, 9, 2))
print(lst)
```

The list is shown below:

```
[4, 6, 8]
```

If we are not using the list() function and using range() alone to create a list, then we will have only range class object returned by the range() function. For example,

```
lst = range(4, 9, 2)
print(lst)
```

The preceding statements will give the following output:

```
range(4, 9, 2)   # this is not a list, it is range object
```

In this case, using a loop like for or while is necessary to view the elements of the list. For example,

```
for i in lst:
    print (i)
```

will display 4, 6, 8. In Program 2, we are showing examples of how to create lists using the range() function. In this program, we are not using the list() function.

Program

Program 2: A Python program to create lists using range() function.

```
# creating lists using range() function
# create a list with 0 to 9 consecutive integer numbers
```

```
list1 = range(10)
for i in list1:  # display element by element
    print(i, ', ', end='')
print()  # throw cursor to next line

#create list with integers from 5 to 9
list2 = range(5, 10)
for i in list2:
    print(i, ', ', end='')
print()

# create a list with odd numbers from 5 to 9
list3 = range(5, 10, 2)  # step size is 2
for i in list3:
    print(i, ', ', end='')
```

Output:

```
C:\>python lists.py
0 , 1 , 2 , 3 , 4 , 5 , 6 , 7 , 8 , 9 ,
5 , 6 , 7 , 8 , 9 ,
5 , 7 , 9 ,
```

We can use while loop or for loop to access elements from a list. The len() function is useful to know the number of elements in the list. For example, len(list) gives total number of elements in the list. The following while loop retrieves starting from 0th to the last element of the list:

```
i=0
while i<len(list):  # repeat from 0 to length of list
    print(list[i])
    i=i+1
```

Observe the len(list) function in the while condition as: while i<len(list). This will return the total number of elements in the list. If the total number of elements is 'n', then the condition will become: while(i<n). It means i values are changing from 0 to n-1. Thus this loop will display all elements of the list from 0 to n-1.

Another way to display elements of a list is by using a for loop, as:

```
for i in list:  # repeat for all elements
    print(i)
```

Here, 'i' will assume one element at a time from the list and hence if we display 'i' value, it will display the elements one by one. In Program 3, we are showing how to access the elements of a list using a while loop and a for loop.

Program

Program 3: A Python program to access list elements using loops.

```
# displaying list elements using while and for loops
list = [10,20,30,40,50]

print('Using while loop')
i=0
while i<len(list):  # repeat from 0 to length of list
    print(list[i])
    i=i+1
```

```
print('Using for loop')
for i in list:  # repeat for all elements
    print(i)
```

Output:

```
C:\>python lists.py
Using while loop
10
20
30
40
50
Using for loop
10
20
30
40
50
```

Updating the Elements of a List

Lists are *mutable*. It means we can modify the contents of a list. We can append, update or delete the elements of a list depending upon our requirements.

Appending an element means adding an element at the end of the list. To append a new element to the list, we should use the append() method. In the following example, we are creating a list with elements from 1 to 4 and then appending a new element 9.

```
lst = list(range(1,5))   # create a list using list() and range()
print(lst)
```

The preceding statements will give the following output:

```
[1, 2, 3, 4]
```

Now, consider the following statements:

```
lst.append(9)   # append a new element to lst
print(lst)
```

The preceding statements will give the following output:

```
[1, 2, 3, 4, 9]
```

Updating an element means changing the value of the element in the list. This can be done by accessing the specific element using indexing or slicing and assigning a new value. Consider the following statements:

```
lst[1]= 8   # update 1st element of lst
print(lst)
```

The preceding statements will give:

```
[1, 8, 3, 4, 9]
```

This logic is shown in Program 4 where we are displaying the elements of a list in reverse order using while loop in two different ways.

Program

Program 4: A Python program to display the elements of a list in reverse order.

```
# displaying list elements in reverse order
days = ['Sunday', 'Monday', 'Tuesday', 'Wednesday', 'Thursday']

print('\nIn reverse order: ')
i=len(days)-1  # i will be 4
while i>=0:
    print(days[i])  # display from 4th to 0th elements
    i-=1

print('\nIn reverse order: ')
i=-1  # days[-1] represents last element
while i>=-len(days):  # display from -1th to -5th elements
    print(days[i])
    i-=1
```

Output:

```
C:\>python lists.py
In reverse order:
Thursday
Wednesday
Tuesday
Monday
Sunday

In reverse order:
Thursday
Wednesday
Tuesday
Monday
Sunday
```

Concatenation of Two Lists

We can simply use '+' operator on two lists to join them. For example, 'x' and 'y' are two lists. If we write x+y, the list 'y' is joined at the end of the list 'x'.

```
x = [10,20,30,40,50]
y = [100,110,120]
print(x+y)    # concatenate x and y
```

The concatenated list appears:

```
[10, 20, 30, 40, 50, 100, 110, 120]
```

Repetition of Lists

We can repeat the elements of a list 'n' number of times using '*' operator. For example, if we write x*n, the list 'x' will be repeated for n times as:

```
print(x*2)   # repeat the list x for 2 times
```

Now, the list appears as:

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

Membership in Lists

We can check if an element is a member of a list or not by using 'in' and 'not in' operator. If the element is a member of the list, then 'in' operator returns True else False. If the element is not in the list, then 'not in' operator returns True else False. See the examples below:

```
x = [10,20,30,40,50]
a = 20
print(a in x)   # check if a is member of x
```

The preceding statements will give:

```
True
```

If you write,

```
print(a not in x)   # check if a is not a member of x
```

Then, it will give

```
False
```

Aliasing and Cloning Lists

Giving a new name to an existing list is called 'aliasing'. The new name is called 'alias name'. For example, take a list 'x' with 5 elements as

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

To provide a new name to this list, we can simply use assignment operator as:

```
y = x
```

In this case, we are having only one list of elements but with two different names 'x' and 'y'. Here, 'x' is the original name and 'y' is the alias name for the same list. Hence, any modifications done to 'x' will also modify 'y' and vice versa. Observe the following statements where an element x[1] is modified with a new value. This is shown in Figure 10.1.

```
x = [10,20,30,40,50]
y = x   # x is aliased as y
print(x)  will display [10,20,30,40,50]
print(y) will display [10,20,30,40,50]
```

```
x[1] = 99   # modify 1st element in x
print(x)  will display [10,99,30,40,50]
print(y) will display [10,99,30,40,50]
```

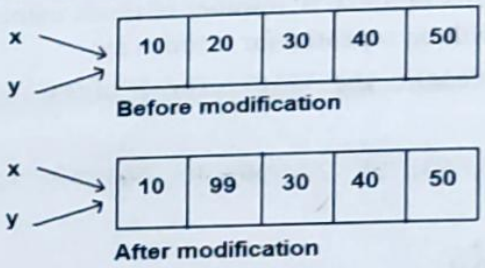

Figure 10.1: Effect of modifications in aliasing

Hence, if the programmer wants two independent lists, he should not go for aliasing. On the other hand, he should use cloning or copying.

Obtaining exact copy of an existing object (or list) is called 'cloning'. To clone a list, we can take help of the slicing operation as:

```
y = x[:]   # x is cloned as y
```

When we clone a list like this, a separate copy of all the elements is stored into 'y'. The lists 'x' and 'y' are independent lists. Hence, any modifications to 'x' will not affect 'y' and vice versa. Consider the following statements:

```
x = [10,20,30,40,50]
y = x[:]   # x is cloned as y
print(x)  will display [10,20,30,40,50]
print(y) will display [10,20,30,40,50]

x[1] = 99   # modify 1st element in x
print(x)  will display [10,99,30,40,50]
print(y) will display [10,20,30,40,50]
```

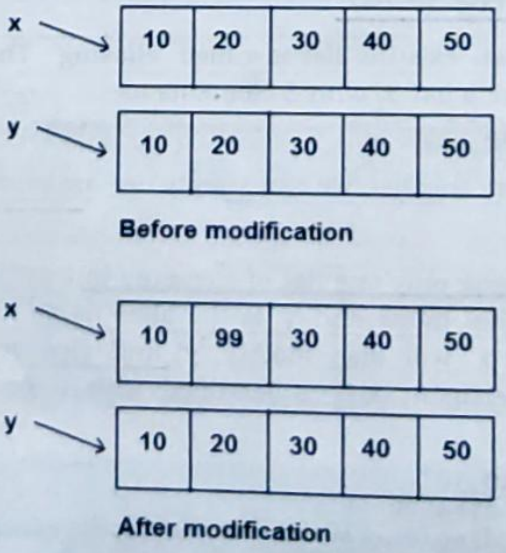

Figure 10.2: Effect of modifications in cloning or copying

We can observe that in cloning, modifications to a list are confined only to that list. The same can be achieved by copying the elements of one list to another using copy() method. For example, consider the following statement:

```
y = x.copy()   # x is copied as y
```

When we copy a list like this, a separate copy of all the elements is stored into 'y'. The lists 'x' and 'y' are independent. Hence, any modifications to 'x' will not affect 'y' and vice versa. Figure 2 depicts the concept of cloning and copying.

Methods to Process Lists

The function len() is useful to find the number of elements in a list. We can use this function as:

```
n = len(list)
```

Here, 'n' indicates the number of elements of the list. Similar to len() function, we have max() function that returns biggest element in the list. Also, the min() function returns the smallest element in the list. Other than these function, there are various other methods provided by Python to perform various operations on lists. These methods are shown in Table 10.1:

Table 10.1: The List methods and their Description

Method	Example	Description
sum()	list.sum()	Returns sum of all elements in the list.
index()	list.index(x)	Returns the first occurrence of x in the list
append()	list.append(x)	Appends x at the end of the list
insert()	list.insert(i, x)	Inserts x in to the list in the position specified by i
copy()	list.copy()	Copies all the list elements into a new list and returns it
extend()	list.extend(list1)	Appends list1 to list
count()	list.count(x)	Returns number of occurrences of x in the list
remove()	list.remove(x)	Removes x from the list
pop()	list.pop()	Removes the ending element from the list
sort()	list.sort()	Sorts the elements of the list into ascending order
reverse()	list.reverse()	Reverses the sequence of elements in the list
clear()	list.clear()	Deletes all elements from the list

```
    x.append(int(input()))  # add the element to the list x
print('The list is: ', x)  # display the list

y = int(input('Enter element to count: '))
c=0
for i in x:
    if(y==i): c+=1
print('{} is found {} times.'.format(y, c))
```

Output:

```
C:\>python lists.py
How many elements? 5
Enter element: 40
Enter element: 20
Enter element: 30
Enter element: 40
Enter element: 50
The list is: [40, 20, 30, 40, 50]
Enter element to count: 40
40 is found 2 times.
```

Finding Common Elements in Two Lists

Sometimes, it is useful to know which elements are repeated in two lists. For example, there is a scholarship for which a group of students enrolled in a college. There is another scholarship for which another group of students got enrolled. Now, we want to know the names of the students who enrolled for both the scholarships so that we can restrict them to take only one scholarship. That means, we are supposed to find out the common students (or elements) in both the lists.

Let's take the two groups of students as two lists. First of all, we should convert the lists into sets, using set() function, as: set(list). Then we should find the common elements in the two sets using intersection() method as:

```
set1.intersection(set2)
```

This method returns a set that contains common or repeated elements in the two sets. This gives us the names of the students who are found in both the sets. This logic is presented in Program 9.

Program

Program 9: A Python program to find common elements in two lists.

```
# finding common elements in two lists

# take two lists
scholar1 = ['Vinay', 'Krishna', 'Saraswathi', 'Govind']
scholar2 = ['Rosy', 'Govind', 'Tanushri', 'Vinay', 'Vishal']

# convert them into sets
s1 = set(scholar1)
s2 = set(scholar2)
```

```
# find intersection of two sets
s3 = s1.intersection(s2)

# convert the resultant set into a list
common = list(s3)
# display the list
print(common)
```

Output:

```
C:\>python lists.py
['Vinay', 'Govind']
```

Storing Different Types of Data in a List

The beauty of lists is that they can store different types of elements. For example, we ca store an integer, a string a float type number etc. in the same string. It is also possible retrieve the elements from the list. This has advantage for lists over arrays. We can crea list 'emp' as an empty list as:

```
emp = []
```

Then we can store employee data like id number, name and salary details into the 'em list using the append() method. After that, it is possible to retrieve employee detai depending on id number of the employee. This is shown in Program 10.

Program

Program 10: A Python program to create a list with employee data and then retrieve particular employee details.

```
# retrieving employee details from a list
emp = []  # take an empty list

n = int(input('How many employees? ')) # accept input into n

for i in range(n):  # repeat for n times
    print('Enter id: ', end='')
    emp.append(int(input()))
    print('Enter name: ', end='')
    emp.append(input())
    print('Enter salary: ', end='')
    emp.append(float(input()))

print('The list is created with employee data.')

id = int(input('Enter employee id: '))

# display employee details upon taking id.
for i in range(len(emp)):
    if id==emp[i]:
        print('Id= {:d}, Name= {:s}, Salary= {:.2f}'.format(emp[i],
    emp[i+1], emp[i+2]))
        break
```

Output:

```
C:\>python lists.py
How many employees? 4
Enter id: 10
Enter name: Vijaya lakshmi
Enter salary: 7000.50
Enter id: 11
Enter name: Gouri shankar
Enter salary: 9500.50
Enter id: 12
Enter name: Anil kumar
Enter salary: 8000
Enter id: 13
Enter name: Hema chandra
Enter salary: 8500.75
The list is created with employee data.
Enter employee id: 12
Id= 12, Name= Anil kumar, Salary= 8000.00
```

Nested Lists

A list within another list is called a *nested* list. We know that a list contains several elements. When we take a list as an element in another list, then that list is called a nested list. For example, we have two lists 'a' and 'b' as:

```
a = [80, 90]
b = [10, 20, 30, a]
```

Observe that the list 'a' is inserted as an element in the list 'b' and hence 'a' is called a nested list. Let's display the elements of 'b', by writing the following statement:

```
print(b)
```

The elements of b appears:

```
[10, 20, 30, [80, 90]]
```

The last element [80, 90] represents a nested list. So, 'b' has 4 elements and they are:

```
b[0] = 10
b[1] = 20
b[2] = 30
b[3] = [80, 90]
```

So, b[3] represents the nested list and if we want to display its elements separately, we can use a for loop as:

```
for x in b[3]:
    print(x)
```

Program

Program 11: A Python program to create a nested list and display its elements.

```
# To create a list with another list as element
list = [10, 20, 30, [80, 90]]
print('Total list= ', list)  # display entire list
print('First element= ', list[0])  # display first element
```

```
print('Last element is nested list= ', list[3])  # display nested list
for x in list[3]:  # display all elements in nested list
    print(x)
```

Output:

```
C:\>python lists.py
Total list=  [10, 20, 30, [80, 90]]
First element=  10
Last element is nested list=  [80, 90]
80
90
```

In Program 11, we used a for loop to display the nested list. We can refer to the individual elements of the nested list, as:

```
list[3][0]   # represents 80
list[3][1]   # represents 90
```

One of the main uses of nested lists is that they can be used to represent matrices. A matrix represents a group of elements arranged in several rows and columns. If a matrix contains 'm' rows and 'n' columns, then it is called m x n matrix. In Python, matrices are created as 2D arrays or using matrix object in numpy. We can also create a matrix using nested lists.

Nested Lists as Matrices

Suppose we want to create a matrix with 3 rows and 3 columns, we should create a list with 3 other lists as:

```
mat = [[1,2,3], [4,5,6], [7,8,9]]
```

Here, 'mat' is a list that contains 3 lists which are rows of the 'mat' list. Each row contains again 3 elements as:

```
[[1,2,3],    # first row
 [4,5,6],    # second row
 [7,8,9]]    # third row
```

If we use a for loop to retrieve the elements from 'mat', it will retrieve row by row, as:

```
for r in mat:
    print(r)   # display row by row
```

But we want to retrieve columns (or elements) in each row; hence we need another for loop inside the previous loop, as:

```
for r in mat:
    for c in r:   # display columns in each row
        print(c, end=' ')
    print()
```

Tuples

A tuple is a Python sequence which stores a group of elements or items. Tuples are similar to lists but the main difference is tuples are immutable whereas lists are mutable. Since tuples are immutable, once we create a tuple we cannot modify its elements. Hence we cannot perform operations like append(), extend(), insert(), remove(), pop() and clear() on tuples. Tuples are generally used to store data which should not be modified and retrieve that data on demand.

Creating Tuples

We can create a tuple by writing elements separated by commas inside parentheses (). The elements can be of same datatype or different types. For example, to create an empty tuple, we can simply write empty parenthesis, as:

```
tup1 = ()  # empty touple
```

If we want to create a tuple with only one element, we can mention that element in parentheses and after that a comma is needed, as:

```
tup2 = (10,)  # tuple with one element. Observe comma after the element.
```

Here is a tuple with different types of elements:

```
tup3 = (10, 20, -30.1, 40.5, 'Hyderabad', 'New Delhi')
```

We can create a tuple with only one of type of elements also, like the following:

```
tup4 = (10, 20, 30)  # tuple with integers
```

If we do not mention any brackets and write the elements separating them by commas, then they are taken by default as a tuple. See the following example:

```
tup5 = 1, 2, 3, 4  #no braces
```

The point to remember is that if do not use any brackets, it will become a tuple and not a list or any other datatype.

It is also possible to create a tuple from a list. This is done by converting a list into a tuple using the tuple() function. Consider the following example:

```
list = [1, 2, 3]   # take a list
tpl = tuple(list)   # convert list into tuple
print(tpl)   # display tuple
```

The tuple is shown below:

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

Another way to create a tuple is by using range() function that returns a sequence. To create a tuple 'tpl' that contains numbers from 4 to 8 in steps of 2, we can use the range() function along with tuple() function, as:

```
tpl= tuple(range(4, 9, 2)) # numbers from 4 to 8 in steps of 2
print(tpl)
```

The preceding statements will give:

```
(4, 6, 8)
```

Accessing the Tuple Elements

Accessing the elements from a tuple can be done using indexing or slicing. This is same as that of a list. For example, let's take a tuple by the name 'tup' as:

```
tup = (50,60,70,80,90,100)
```

Indexing represents the position number of the element in the tuple. Now, tup[0] represents the 0th element, tup[1] represents the 1st element and so on.

```
print(tup[0])
```

The preceding will give:

```
50
```

Now, if you write:

```
print(tup [5])
```

Then the following element appears:

```
100
```

Similarly, negative indexing is also possible. For example, tup[-1] indicates the last element and tup[-2] indicates the second element from the end and so on. Consider the following statement:

```
print(tup[-1])
```

The preceding statement will give:

```
100
```

If you write,

```
print(tup[-6])
```

Then the following output appears:

```
50
```

Slicing represents extracting a piece or part of the tuple. Slicing is done in the format: [start: stop: stepsize]. Here, 'start' represents the position of the starting element and 'stop' represents the position of the ending element and the 'stepsize' indicates the incrementation. If the tuple contains 'n' elements, the default values will be 0 for 'start' and n-1 for 'stop' and 1 for 'stepsize'. For example, to extract all the elements from the tuple, we can write:

```
print(tup[:])
```

The elements of tuple appear:

```
(50, 60, 70, 80, 90, 100)
```

For example, to extract the elements from 1st to 4th, we can write:

```
print(tup[1:4])
```

The elements of tuple appear:

```
(60, 70, 80)
```

Similarly, to extract every other element, i.e. alternate elements, we can write:

```
print(tup[::2])   # here, start and stop assume default values.
```

The following output appears:

```
(50, 70, 90)
```

Negative values can be given in the slicing. If the 'step size' is negative, the elements are extracted in reverse order as:

```
print(tup[::-2])
```

The elements appear in the reverse order:

```
(100, 80, 60)
```

When the step size is not negative, the elements are extracted from left to right. In the following example, starting position -4 indicates the 4th element from the last. Ending position -1 indicates the last element. Hence, the elements from 4th to one element before the ending element (left to right) will be extracted.

```
print(tup[-4:-1])   # here, step size is 1
```

The following elements appear:

```
(70, 80, 90)
```

In most of the cases, the extracted elements from the tuple should be stored in separate variables for further use. In the following example, we are extracting the first two elements from the 'student' tuple and storing them into two variables.

```
student = (10, 'vinay kumar', 50,60,65,61,70)
rno, name = student[0:2]
```

Now, the variable 'rno' represents 10 and 'name' represents Vinaykumar. Consider the following statement:

```
print(rno)
```

The preceding statement will give:

```
10
```

Now, if you write:

```
print(name)
```

The preceding statement will provide the name of the student:

```
Vinay kumar
```

If we want to retrieve the marks of the student from 'student' tuple, we can do it as:

```
marks = student[2:7] # store the elements from 2nd to 6th into 'marks'
# tuple
for i in marks:
    print(i)
```

The preceding statements will give the following output:

```
50
60
65
61
70
```

Basic Operations on Tuples

The 5 basic operations: finding length, concatenation, repetition, membership and iteration operations can be performed on any sequence may be it is a string, list, tuple or a dictionary.

To find length of a tuple, we can use len() function. This returns the number of elements in the tuple.

Consider the following example:

```
student = (10, 'Vinaykumar', 50,60,65,61,70)
len(student)
```

The preceding statement will give the following output:

```
7
```

We can concatenate or join two tuples and store the result in a new tuple. For example, the student paid a fees of Rs. 25,000.00 every year for 4 terms, then we can create a 'fees' tuple as:

```
fees = (25000.00,)*4   # repeat the tuple elements for 4 times.
print(fees)
```

The preceding statement will give the following output:

```
(25000.0, 25000.0, 25000.0, 25000.0)
```

Now, we can concatenate the 'student' and 'fees' tuples together to form a new tuple 'student1' as:

```
student1 = student+fees
print(student1)
```

The preceding statement will give:

```
(10, 'Vinay kumar', 50, 60, 65, 61, 70, 25000.0, 25000.0, 25000.0,
    25000.0)
```

Searching whether an element is a member of the tuple or not can be done using 'in' and 'not in' operators. The 'in' operator returns True if the element is a member. The 'not in' operator returns True if the element is not a member. Consider the following statements:

```
name='Vinay kumar'   # to know if this is member of student1 or not
name in student1
```

The preceding statements will give:

```
True
```

Suppose if you write:

```
name not in student1
```

Then the following output will appear:

```
False
```

The repetition operator repeats the tuple elements. For example, we take a tuple 'tpl' and repeat its elements for 4 times as:

```
tpl = (10, 11, 12)
tpl1 = tpl*3   # repeat for 3 times and store in tpl1
print(tpl1)
```

The preceding statements will give the following output:

```
(10, 11, 12, 10, 11, 12, 10, 11, 12)
```

Functions to Process Tuples

HOME WORK

There are a few functions provided in Python to perform some important operations on tuples. These functions are mentioned in Table 10.2. Any function is called directly with its name. In this table, count() and index() are not functions. They are methods. Hence, they are called in the format: object.method().

Table 10.2: The functions available to process tuples

Function	Example	Description
len()	len(tpl)	Returns the number of elements in the tuple.
min()	min(tpl)	Returns the smallest element in the tuple.
max()	max(tpl)	Returns the biggest element in the tuple.

Function	Example	Description
count()	tpl.count(x)	Returns how many times the element 'x' is found in tpl.
index()	tpl.index(x)	Returns the first occurrence of the element 'x' in tpl. Raises ValueError if 'x' is not found in the tuple.
sorted()	sorted(tpl)	Sorts the elements of the tuple into ascending order. sorted(tpl, reverse=True) will sort in reverse order.

We will write a program to accept elements of a tuple from the keyboard and find their sum and average. In this program, we are using the following statement to accept elements from the keyboard directly in the format of a tuple (i.e. inside parentheses).

```
num = eval(input("Enter elements in (): "))
```

The eval() function is useful to evaluate whether the typed elements are a list or a tuple depending upon the format of brackets given while typing the elements. If we type the elements inside the square braces as: [1,2,3,4,5] then they are considered as elements of a list. If we enter the elements inside parentheses as: (1,2,3,4,5), then they are considered as elements of a tuple. Even if do not type any brackets, then by default they are taken as elements of tuple.

Program

Program 14: A Python program to accept elements in the form of a tuple and display their sum and average.

```
# program to find sum and average of elements in a tuple
num = eval(input("Enter elements in (): "))
sum=0
n=len(num) # n is no. of elements in the tuple
for i in range(n):  # repeat i from 0 to n-1
    sum+=num[i]  # add each element to sum
print('Sum of numbers: ', sum)  # display sum
print('Average of numbers: ', sum/n)  #display average
```

Output:

```
C:\>python tuples.py
Enter elements in (): (1,2,3,4,5,6)
Sum of numbers: 21
Average of numbers: 3.5
```

When the above program asks the user for entering the elements, the user can type the elements inside the parentheses as: (1,2,3,4,5,6) or without any parentheses as: 1,2,3,4,5,6. When a group of elements are entered with commas (,), then they are by default taken as a tuple in Python.

In Program 15, first we enter the elements separated by commas. These elements are stored into a string 'str' as:

```
str = input('Enter elements separated by commas: ').split(',')
```

Then each element of this string is converted into integer and stored into a list 'lst' as:

```
lst = [int(num) for num in str]
```

This list can be converted into a tuple using tuple() function as:

```
tup = tuple(lst)
```

Later, index() method is used to find the first occurrence of the element 'ele' as:

```
pos = tup.index(ele)    # returns first occurrence of element
```

If the 'ele' is not found in the tuple, then there will be an error by the name 'ValueError' which should be handled using try and except blocks which is shown in the program.

Program

Program 15: A Python program to find the first occurrence of an element in a tuple.

```
# inserting elements from keyboard into the tuple and finding element
    # position
# accept elements from keyboard as strings separated by commas
str = input('Enter elements separated by commas: ').split(',')

lst = [int(num) for num in str]   # convert strings into integers and
# store into a list
tup = tuple(lst)   # convert list into tuple

print('The tuple is: ', tup)  # display the tuple

ele = int(input('Enter an element to search: '))
try:
    pos = tup.index(ele)   # returns first occurrence of element
    print('Element position no: ', pos+1)

exceptValueError:  # if element not found, ValueError will rise
    print('Element not found in tuple')
```

Output:

```
C:\>python tuples.py
Enter elements separated by commas: 10,20,30,20,40
The tuple is: (10, 20, 30, 20, 40)
Enter an element to search: 20
Element position no: 2
```

Nested Tuples

A tuple inserted inside another tuple is called nested tuple. For example,

```
tup = (50,60,70,80,90, (200, 201))  # tuple with 6 elements
```

Observe the last parentheses in the tuple 'tup', i.e. (200, 201). These parentheses represent that it is a tuple with 2 elements inserted into the tuple 'tup'. The tuple (200, 201) is called nested tuple as it is inside another tuple.

The nested tuple with the elements (200, 201) is treated as an element along with other elements in the tuple 'tup'. To retrieve the nested tuple, we can access it as an ordinary element as tup[5] as its index is 5. Now, consider the following statement:

```
print('Nested tuple= ', tup[6])
```

The preceding statement will give the following output:

```
Nested tuple=  (200, 201)
```

Every nested tuple can represent a specific data record. For example, to store 4 employees data in 'emp' tuple, we can write:

```
emp = ((10, "Vijay", 9000.90), (20, "Nihaar", 5500.75), (30,
     "Vanaja",8900.00), (40, "Kapoor", 5000.50))
```

Here, 'emp' is the tuple name. It contains 4 nested tuples each of which represents the data of an employee. Every employee's identification number, name and salary are stored as a nested tuples.

Sorting Nested Tuples

To sort a tuple, we can use sorted() function. This function sorts by default into ascending order. For example,

```
print(sorted(emp))
```

will sort the tuple 'emp' in ascending order of the 0th element in the nested tuples, i.e. identificaton number. If we want to sort the tuple based on employee name, which is the 1st element in the nested tuples, we can use a lambda expression as:

```
print(sorted(emp, key=lambda x: x[1]))  # sort on name
```

Here, key indicates the key for the sorted() function that tells on which element sorting should be done. The lambda function: lambda x: x[1] indicates that x[1] should be taken as the key that is nothing but 1st element. If we want to sort the tuple based on salary, we can use the lambda function as: lambda x: x[2]. Consider Program 16.

Program

Program 16: A Python program to sort a tuple with nested tuples.

```
# sorting a tuple that contains tuples as elements
# take employee tuple with id number, name and salary
emp = ((10, "Vijay", 9000.90), (20, "Nihaar", 5500.75), (30,
     "Vanaja",8900.00), (40, "Kapoor", 5000.50))

print(sorted(emp))  # sorts by default on id
print(sorted(emp, reverse=True))  # reverses on id
print(sorted(emp, key=lambda x: x[1]))  # sort on name
print(sorted(emp, key=lambda x: x[2]))  # sort on salary
```

Output:

```
C:\>python tuples.py
[(10, 'Vijay', 9000.9), (20, 'Nihaar', 5500.75), (30, 'Vanaja',
    8900.0), (40, 'Kapoor', 5000.5)]
[(40, 'Kapoor', 5000.5), (30, 'Vanaja', 8900.0), (20, 'Nihaar',
    5500.75), (10, 'Vijay', 9000.9)]
[(40, 'Kapoor', 5000.5), (20, 'Nihaar', 5500.75), (30, 'Vanaja',
    8900.0), (10, 'Vijay', 9000.9)]
[(40, 'Kapoor', 5000.5), (20, 'Nihaar', 5500.75), (30, 'Vanaja',
    8900.0), (10, 'Vijay', 9000.9)]
```

Inserting Elements in a Tuple

Since tuples are immutable, we cannot modify the elements of the tuple once it is created. Now, let's see how to insert a new element into an existing tuple. Let'stake 'x' as an existing tuple. Since 'x' cannot be modified, we have to create a new tuple 'y' with the newly inserted element. The following logic can be used:

1. First of all, copy the elements of 'x' from 0th position to pos-2 position into 'y' as:
   ```
   y = x[0:pos-1]
   ```
2. Concatenate the new element to the new tuple 'y'.
   ```
   y = y+new
   ```
3. Concatenate the remaining elements (from pos-1 till end) of x to the new tuple 'y'. The total tuple can be stored again with the old name 'x'.
   ```
   x = y+x[pos-1:]
   ```

This logic is depicted in Figure 10.4 and shown in Program 17:

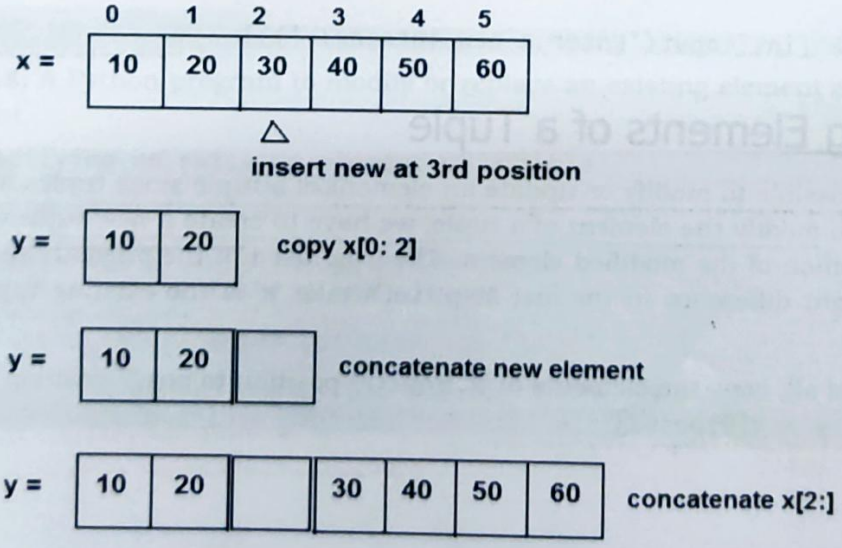

Figure 10.4: To insert a new element into a tuple

Program

Program 17: A Python program to insert a new element into a tuple of elements at a specified position.

```
# inserting a new element into a tuple
names = ('Visnu', 'Anupama', 'Lakshmi', 'Bheeshma')
print(names)

# accept new name and position number
lst= [input('Enter a new name: ')]
new = tuple(lst)
pos = int(input('Enter position no: '))

# copy from 0th to pos-2 into another tuple names1
names1 = names[0:pos-1]

# concatenate new element at pos-1
names1 = names1+new

# concatenate the remaining elements of names from pos-1 till end
names = names1+names[pos-1:]
print(names)
```

Output:

```
C:\>python tuples.py
('Vishnu', 'Anupama', 'Lakshmi', 'Bheeshma')
Enter a new name: Ganesh
Enter position no: 2
('Vishnu','Ganesh', 'Anupama', 'Lakshmi', 'Bheeshma')
```

This program works well with strings. Of course the same program can be used with integers also, if we change the input statement that accepts an integer number as:

```
lst= [int(input('Enter a new integer: '))]
```

Modifying Elements of a Tuple

It is not possible to modify or update an element of a tuple since tuples are immutable. If we want to modify the element of a tuple, we have to create a new tuple with a new value in the position of the modified element. The logic used in the previous section holds good with a slight difference in the last step. Let's take 'x' is the existing tuple and 'y' is the new tuple.

1. First of all, copy the elements of 'x' from 0th position to pos-2 position into 'y' as:

```
y = x[0:pos-1]
```

2. Concatenate the new element to the new tuple 'y'. Thus the new element is stored in the position of the element being modified.

```
y = y+new
```

3. Now concatenate the remaining elements from 'x' by eliminating the element which is at 'pos-1'. It means we should concatenate elements from 'pos' till the end. The total tuple can be assigned again to the old name 'x'.

```
x = y+x[pos:]
```

This logic is depicted in Figure 10.5 and shown in Program 18:

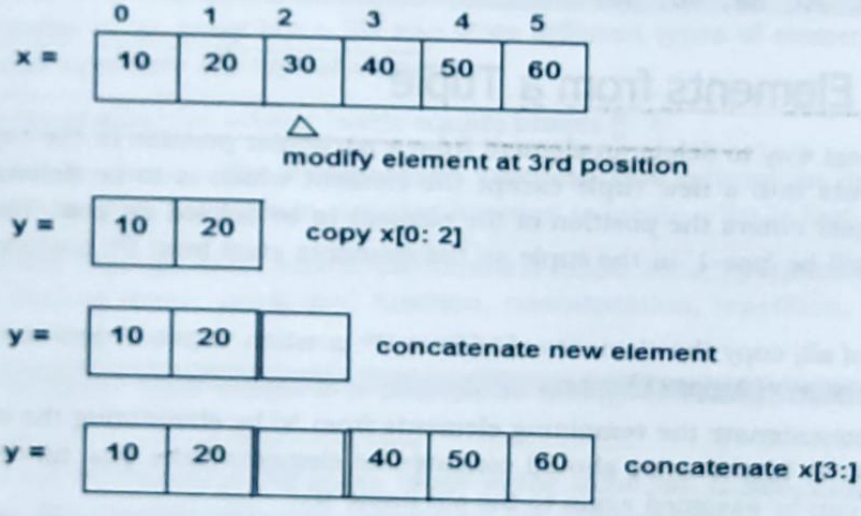

Figure 10.5: To modify a particular element in a tuple

Program

Program 18: A Python program to modify or replace an existing element of a tuple with a new element.

```
# modifying an existing element of a tuple
num = (10, 20, 30, 40, 50)
print(num)

# accept new element and position number
lst= [int(input('Enter a new element: '))]
new = tuple(lst)
pos = int(input('Enter position no: '))

# copy from 0th to pos-2 into another tuple num1
num1 = num[0:pos-1]
```

```
# concatenate new element at pos-1
num1 = num1+new

# concatenate the remaining elements of num from pos till end
num = num1+num[pos:]
print(num)
```

Output:

```
C:\>python tuples.py
(10, 20, 30, 40, 50)
Enter a new element: 88
Enter position no: 3
(10, 20, 88, 40, 50)
```

Deleting Elements from a Tuple

The simplest way to delete an element from a particular position in the tuple is to copy all the elements into a new tuple except the element which is to be deleted. Let's assume that the user enters the position of the element to be deleted as 'pos'. The corresponding position will be 'pos-1' in the tuple as the elements start from 0^{th} position. Now the logic will be:

1. First of all, copy the elements of 'x' from 0^{th} position to pos-2 position into 'y' as:
   ```
   y = x[0:pos-1]
   ```
2. Now concatenate the remaining elements from 'x' by eliminating the element which is at 'pos-1'. It means we should concatenate elements from 'pos' till the end. The total tuple can be assigned again to the old name 'x'.
   ```
   x = y+x[pos:]
   ```

Program

Program 19: A program to delete an element from a particular position in the tuple.

```
# deleting an element of a tuple
num = (10, 20, 30, 40, 50)
print(num)
# accept position number of the element to delete
pos = int(input('Enter position no: '))

# copy from 0th to pos-2 into another tuple num1
num1 = num[0:pos-1]

# concatenate the remaining elements of num from pos till end
num = num1+num[pos:]
print(num)
```

Output:

```
C:\>python tuples.py
(10, 20, 30, 40, 50)
Enter position no: 3
(10, 20, 40, 50)
```

Points to Remember

- A sequence is a datatype that represents a group of elements. In Python, strings, lists, tuples and dictionaries are very important sequence datatypes.
- A list is similar to an array but a list can store different types of elements; whereas, an array can store only one type of elements.
- The elements of a list are written inside square braces [].
- It is possible to create a list using range() function. This returns an iterable object whose elements form a sequence. Another function to create a list is list() function.
- Any sequence (strings, lists, tuples, dictionaries) follow some operations like slicing, indexing, finding length using len() function, concatenation, repetition, membership and iteration operations.
- Lists are mutable. That means it is possible to modify or change the elements of a list.
- Aliasing a list represents giving a new name to the same list. Hence, if we change the aliased list, the changes will affect the original list and vice versa.
- Cloning a list can be done using slice operation. In cloning, if the cloned list is modified, the original list will not be modified as both will represent different copies of lists. The same effect can be seen if we copy the list using copy() method.
- The intersection() method of sets is useful to filter the common elements from two sets.
- A list within another list is called nested list. Nested lists are useful to create matrices.
- List comprehensions represent creation of new lists from an iterable object (like a list, set, tuple, dictionary or range) that satisfies a given condition.
- Tuples are similar to lists with the difference that tuples are immutable, whereas lists are mutable. Since tuples are immutable, once we create a tuple, we cannot modify its elements.

- We can create a tuple by writing elements separated by commas inside parentheses ().
- We can convert a list into a tuple using tuple() function.
- The eval() function is useful to evaluate whether the typed elements are a list or a tuple depending upon the format of brackets given while typing the elements.
- If we want to delete, modify or insert elements of a tuple, since tuples are immutable, we have to create a new tuple and store the updated elements.