UNIT - III

DATA STRUCTURES

A sequence is a data type that represents a group of elements. The purpose of any sequence is to store and process group elements. In python, strings, lists, tuples and dictionaries are very important sequence data types.

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.

Creating a List:

Creating a list is as simple as putting different comma-separated values between square brackets.

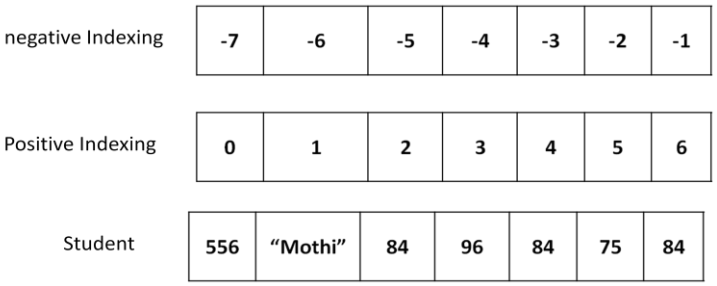
student = [556, “Mothi”, 84, 96, 84, 75, 84]

We can create empty list without any elements by simply writing empty square brackets as: student=[ ]

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 (,).

Accessing Values in list:

To access values in lists, use the square brackets for slicing along with the index or indices to obtain value available at that index. To view the elements of a list as a whole, we can simply pass the list name to print function.



Ex:

```
student =
[556,
“Mothi”,
84, 96, 84,
75, 84 ]
print
student
print student[0] # Access 0th element
print student[0:2] # Access 0th to 1st elements
print student[2: ] #
Access 2nd to end of
list elements print
student[ :3] #
```

Access starting to
2nd elements print
student[:] #
Access starting to
ending elements
print student[-1] #
Access last index
value
print student[-1:-7:-1] # Access elements in
reverse order

Output:

```
[556, “Mothi”, 84, 96, 84, 75, 84]
Mothi
[556, “Mothi”]
[84, 96, 84, 75, 84]
[556, “Mothi”, 84]
[556, “Mothi”, 84, 96, 84, 75, 84]
84
[84, 75, 84, 96, 84, “Mothi”]
```

Creating lists using range() function:

We can use range() function to generate a sequence of integers which can be stored in a list. To store numbers from 0 to 10 in a list as follows.

```
numbers = list( range(0,11) )
print numbers # [0,1,2,3,4,5,6,7,8,9,10]
```

To store even numbers from 0 to 10in a list as follows.

```
numbers = list( range(0,11,2) )
print numbers # [0,2,4,6,8,10]
```

Looping on lists:

We can also display list by using for loop (or) while loop. The len( ) function useful to know the numbers of elements in the list. while loop retrieves starting from 0^{th} to the last element i.e. n-1

Ex-1:

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

Output:

```
1 2 3 4 5
```

Example

```
# creating an empty list
lst = []
 # number of elements as input
 n = int(input("Enter number of elements : "))
 # iterating till the range
 for i in range(0, n):
     ele = int(input())
 # adding the element
 lst.append(ele)
 print(lst)
```

Updating and deleting lists:

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.

Example:

```
lst=[1,2,4,5,8,6]
print lst        # [1,2,4,5,8,6]
lst.append(9)
print lst        # [1,2,4,5,8,6,9]
```

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.

Example:

```
lst=[4,7,6,8,9,3]
print lst        # [4,7,6,8,9,3]
lst[2]=5         # updates 2nd element in the list
print lst        # [4,7,5,8,9,3]
lst[2:5]=10,11,12       # update 2nd element to 4th element in the list
print lst        # [4,7,10,11,12,3]
```

Deleting an element from the list can be done using *'del'* statement. The *del* statement takes the position number of the element to be deleted.

Example:

```
lst=[5,7,1,8,9,6]
del lst[3]       # delete 3rd element from the list i.e., 8
print lst        # [5,7,1,9,6]
```

If we want to delete entire list, we can give statement like *del lst.*

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".

Example:

```
x=[10,20,32,15,16]
y=[45,18,78,14,86]
print x+y        # [10,20,32,15,16,45,18,78,14,86]
```

Repetition of Lists:

We can repeat the elements of a list "n" number of times using "*" operator.

```
x=[10,54,87,96,45]
print x*2        # [10,54,87,96,45,10,54,87,96,45]
```

Membership in Lists:

We can check if an element is a member of a list by using "in" and "not in" operator. If the element is a member of the list, then "in" operator returns **True** otherwise returns **False**. If the element is not in the list, then "not in" operator returns **True** otherwise returns **False**.

Example:

```
x=[10,20,30,45,55,65]
a=20
print a in x      # True
```

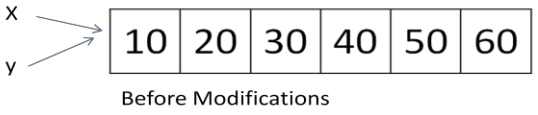
```
a=25
print a in x      #
False a=45
print a not in x # False
a=40
print a not in x # True
```

Aliasing and Cloning Lists:

Giving a new name to an existing list is called *'aliasing'*. The new name is called *'alias name'*. To provide a new name to this list, we can simply use assignment operator (=).

Example:

```
x = [10, 20, 30, 40, 50, 60]
y=x             # x is aliased as y
print x         # [10,20,30,40,50,60]
print y         # [10,20,30,40,50,60]
x[1]=90         # modify 1st element in x
print x         # [10,90,30,40,50,60]
print y         # [10,90,30,40,50,60]
```



Before Modifications

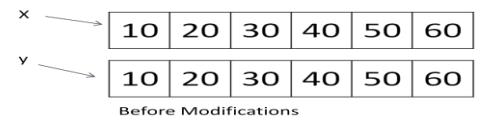
After Modifications

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.

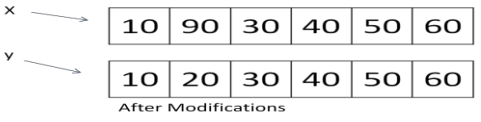
Obtaining exact copy of an existing object (or list) is called "*cloning*". To Clone a list, we can take help of the slicing operation [:].

Example:

```
x = [10, 20, 30, 40, 50, 60]
y=x[:]          # x is cloned as y
print x         # [10,20,30,40,50,60]
print y         # [10,20,30,40,50,60]
x[1]=90         # modify 1st element in x
print x         # [10,90,30,40,50,60]
print y         # [10,20,30,40,50,60]
```



Before Modifications



After Modifications

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.

Methods in Lists:

Method	Description
lst.index(x)	Returns the first occurrence of x in the list.
lst.append(x)	Appends x at the end of the list.
lst.insert(i,x)	Inserts x to the list in the position specified by i.
lst.copy()	Copies all the list elements into a new list and returns it.
lst.extend(lst2)	Appends lst2 to list.
lst.count(x)	Returns number of occurrences of x in the list.
lst.remove(x)	Removes x from the list.
lst.pop()	Removes the ending element from the list.
lst.sort()	Sorts the elements of list into ascending order.
lst.reverse()	Reverses the sequence of elements in the list.
lst.clear()	Deletes all elements from the list.
max(lst)	Returns biggest element in the list.
min(lst)	Returns smallest element in the list.

Example:

```
lst=[10,25,45,51,45,51,21,65]
lst.insert(1,46)
print lst          # [10,46,25,45,51,45,51,21,65]
print lst.count(45)      # 2
```

Finding Common Elements in 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) both the 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.

Example:

```
scholar1=[ "mothi", "sudheer", "vinay", "narendra",
"ramakoteswararao"]
]scholar2=[ "vinay", "narendra", "ramesh"]
s1=set(scholar1)
s2=set(scholar2)
s3=s1.intersection(s2)
common =list(s3)
print common          # display [ "vinay", "narendra" ]
```

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.

Example:

```
a=[10,20,30]
b=[45,65,a]
print b          # display [ 45, 65, [ 10, 20, 30 ] ]
print b[1]       # display 65
print b[2]       # display [ 10, 20, 30 ]
print b[2][0]    # display 10
for x in b[2]:
        print  x,        # display 10 20 30
```

Nested Lists as Matrices:

Suppose we want to create a matrix with 3 rows 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
```

One of the main use 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. In python, matrices are created as 2D arrays or using matrix object in numpy. We can also create a matrix using nested lists.

Q) Write a program to perform addition of two matrices.

```
a=[[1,2,3],[4,5,6],[7,8,9]]
b=[[4,5,6],[7,8,9],[1,2,3]]
c=[[0,0,0],[0,0,0],[0,0,0]]
m1=len(a)
n1=len(a[0])
m2=len(b)
n2=len(b[0])
for i in range(0,m1):
    for j in range(0,n1):
        c[i][j]= a[i][j]+b[i][j]
```

```
5    7    9
11   13   15
8    10   12
```

Q) Write a program to perform multiplication of two matrices. a=[[1,2,3],[4,5,6]]
b=[[4,5],[7,8],[1,2]]
c=[[0,0],[0,0]]

```
m1=len(a)
n1=len(a[0])
m2=len(b)
n2=len(b[0])
for i in range(0,m1):
    for j in range(0,n2):
        for k in range(0,n1):
            c[i][j] += a[i][k]*b[k][j]
for i in range(0,m1):
    for j in range(0,n2):
        print "\t",c[i][j],
    print " "
```

```
21      27
57      72
```

TUPLE:

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. 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 same data type or different types.

To create an empty tuple, we can simply write empty parenthesis, as:

```
tup=( )
```

To create a tuple with only one element, we can, mention that element in parenthesis and after that a comma is needed. In the absence of comma, python treats the element assign ordinary data type.

```
tup = (10)
print tup          # display 10
print type(tup)    # display <type “int”>
```

```
tup = (10,)
print tup          # display 10
print type(tup)    # display<type “tuple”>
```

To create a tuple with different types of elements:

```
tup=(10, 20, 31.5, „Gudivada“)
```

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

```
tup= 10, 20, 34, 47
```

It is possible to create a tuple from a list. This is done by converting a list into a tuple using tuple function.

```
n=[1,2,3,4]
tp=tuple(n)
print tp          # display (1,2,3,4)
```

Another way to create a tuple by using range( ) function that returns a sequence.

```
t=tuple(range(2,11,2))
print t           # display (2,4,6,8,10)
```

Accessing the tuple elements:

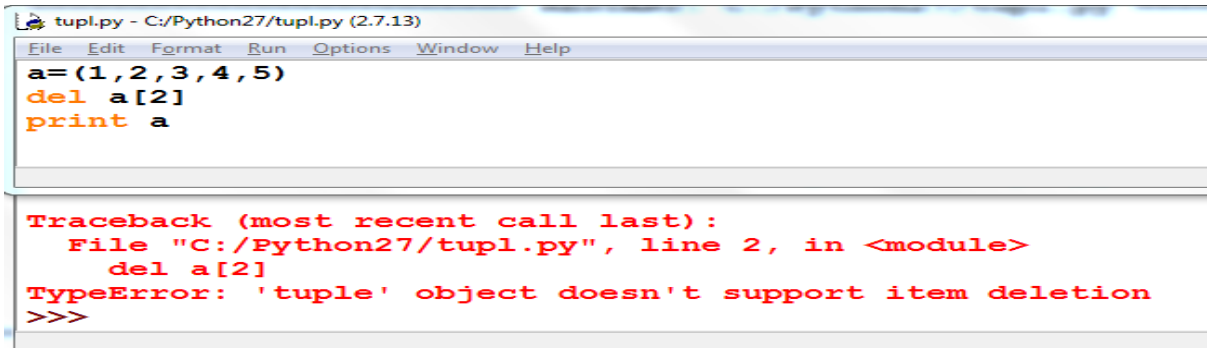
Accessing the elements from a tuple can be done using indexing or slicing. This is same as that of a list. Indexing represents the position number of the element in the tuple. The position starts from 0.

```
tup=(50,60,70,80,90)
print tup[0]          # display 50
print tup[1:4]        # display (60,70,80)
print tup[-1]         # display 90
print tup[-1:-4:-1]   # display (90,80,70)
print tup[-4:-1]      # display (60,70,80)
```

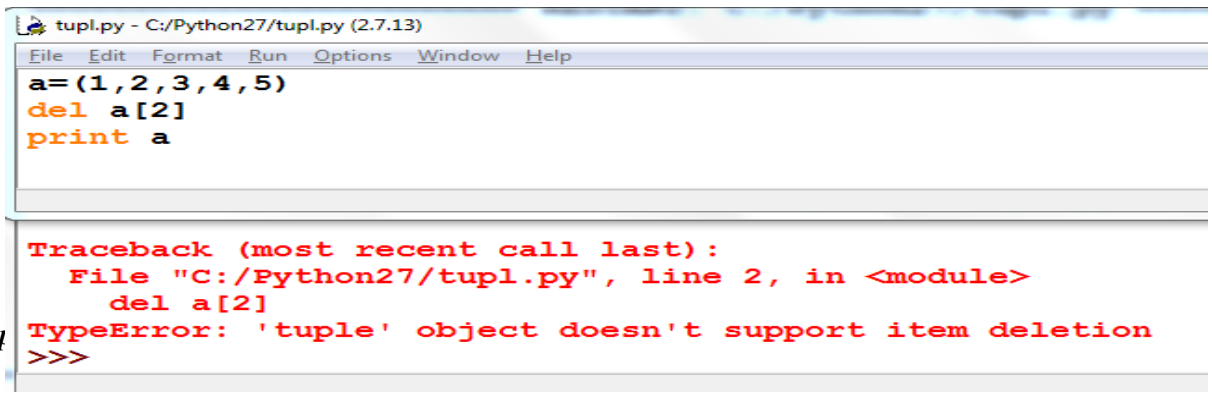
Updating and deleting elements:

Tuples are immutable which means you cannot update, change or delete the values of tuple elements.

Example-1:

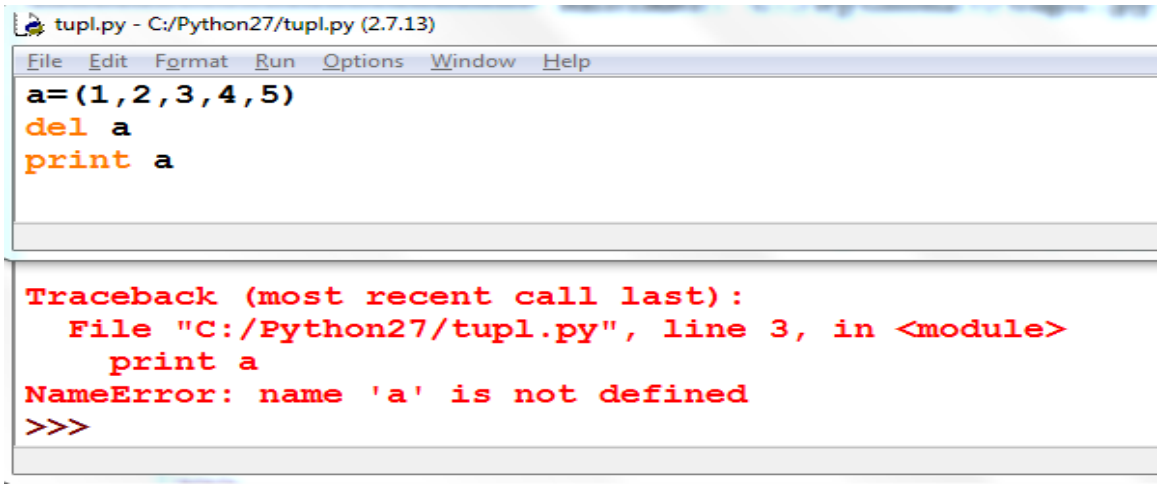


Example 2



Del

However, you can always delete the entire tuple by using the statement.



Note that this exception is raised because we are trying to print the deleted element.

Operations on tuple:

Operation	Description
len(t)	Return the length of tuple.
tup1+tup2	Concatenation of two tuples.
Tup*n	Repetition of tuple values in n number of times.
x in tup	Return True if x is found in tuple otherwise returns False.
cmp(tup1,tup2)	Compare elements of both tuples
max(tup)	Returns the maximum value in tuple.
min(tup)	Returns the minimum value in tuple.
tuple(list)	Convert list into tuple.
tup.count(x)	Returns how many times the element „x" is found in tuple.
tup.index(x)	Returns the first occurrence of the element „x" in tuple. Raises ValueError if „x" is not found in the tuple.
sorted(tup)	Sorts the elements of tuple into ascending order. sorted(tup,reverse=True) will sort in reverse order.

cmp(tuple1, tuple2)

The method **cmp()** compares elements of two tuples.

Syntax

cmp(tuple1, tuple2)

Parameters

tuple1 -- This is the first tuple to be compared

tuple2 -- This is the second tuple to be compared

Return Value

If elements are of the same type, perform the compare and return the result. If elements are different types, check to see if they are numbers.

- If numbers, perform numeric coercion if necessary and compare.

- If either element is a number, then the other element is "larger" (numbers are "smallest").
- Otherwise, types are sorted alphabetically by name.

If we reached the end of one of the tuples, the longer tuple is "larger." If we exhaust both tuples and share the same data, the result is a tie, meaning that 0 is returned.

Example:

```
tuple1 = (123, 'xyz')
tuple2 = (456, 'abc')
print cmp(tuple1,          #display
tuple2) print              -1
cmp(tuple2, tuple1)        #display
                           1
```

Nested Tuples:

Python allows you to define a tuple inside another tuple. This is called a *nested tuple*.

```
students=((“RAVI”, “CSE”, 92.00), (“RAMU”, “ECE”, 93.00), (“RAJA”, “EEE”, 87.00))
for i in students:
print i
```

Output: (“RAVI”, “CSE”, 92.00)
(“RAMU”, “ECE”, 93.00)
(“RAJA”, “EEE”, 87.00)

SET:

Set is another data structure supported by python. Basically, sets are same as lists but with a difference that sets are lists with no duplicate entries. Technically a set is a mutable and an unordered collection of items. This means that we can easily add or remove items from it.

Creating a Set:

A set is created by placing all the elements inside curly brackets {}. Separated by comma or by using the built-in function set( ).

Syntax:

```
Set_variable_name={var1, var2, var3, var4, …….}
```

Example:

```
s={1, 2.5, “abc” }
print s          # display set( [ 1, 2.5, “abc” ] )
```

Converting a list into set:

A set can have any number of items and they may be of different data types. *set()* function is used to converting list into set.

```
s=set( [ 1, 2.5, “abc” ] )
print s          # display set( [ 1, 2.5, “abc” ] )
```

We can also convert tuple or string into set.

```
tup= ( 1, 2, 3, 4, 5 )
print set(tup) # set( [ 1, 2, 3, 4, 5 ] )
str= “MOTHILAL”
print str          # set( [ 'i', 'h', 'm', 't', 'o' ] )
```

Operations on set:

Sno	Operation	Result
1	`len(s)`	number of elements in set *s* (cardinality)
2	`x in s`	test *x* for membership in *s*
3	`x not in s`	test *x* for non-membership in *s*
4	`s.issubset(t)` (or) `s <= t`	test whether every element in *s* is in *t*
5	`s.issuperset(t)` (or) `s >= t`	test whether every element in *t* is in *s*
6	`s = = t`	Returns True if two sets are equivalent and returns False.
7	`s ! = t`	Returns True if two sets are not equivalent and returns False.
8	`s.union(t)` (or) `s|t`	new set with elements from both *s* and *t*
9	`s.intersection(t)` (or) `s & t`	new set with elements common to *s* and *t*

Sno	Operation	Result
10	`s.difference(t)` (or) `s-t`	new set with elements in *s* but not in *t*
11	`s.symmetric_difference(t)` (or) `s ^ t`	new set with elements in either *s* or *t* but not both
12	`s.copy()`	new set with a shallow copy of *s*
13	`s.update(t)`	return set s with elements added from t
14	`s.intersection_update(t)`	return set s keeping only elements also found in t
15	`s.difference_update(t)`	return set s after removing elements found in t
16	`s.symmetric_difference_update(t)`	return set s with elements from s or t but not both
17	`s.add(x)`	add element x to set s
18	`s.remove(x)`	remove x from set s; raises KeyError if not present
19	`s.discard(x)`	removes x from set s if present
20	`s.pop()`	remove and return an arbitrary element from s; raises KeyError if empty
21	`s.clear()`	remove all elements from set s
22	`max(s)`	Returns Maximum value in a set
23	`min(s)`	Returns Minimum value in a set
24	`sorted(s)`	Return a new sorted list from the elements in the set.

To create an empty set you cannot write s={}, because python will make this as a directory. Therefore, to create an empty set use *set( )* function.

```
s=set( )                                    s={}
print type(s)   # display <type „set“>      print type(s) # display <type „dict“>
```

Updating a set:

Since sets are unordered, indexing has no meaning. Set operations do not allow users to access or change an element using indexing or slicing.

Dictionary:

A dictionary represents a group of elements arranged in the form of key-value pairs. The first element is considered as “key” and the immediate next element is taken as its
“value”. The key and its value are separated by a colon (:). All the key-value pairs in a dictionary are inserted in curly braces { }.

```
d= { “Regd.No”: 556, “Name”:”Mothi”, “Branch”: “CSE” }
```

Here, the name of dictionary is “dict”. The first element in the dictionary is a string “Regd.No”. So, this is called “key”. The second element is 556 which is taken as its “value”.

Example:

```
d= { “Regd.No”: 556, “Name”:”Mothi”, “Branch”:
“CSE”} print d[“Regd.No”]       # 556
print d[“Name”]   # Mothi
print d[“Branch”] # CSE
```

To access the elements of a dictionary, we should not use indexing or slicing. For example, dict[0] or dict[1:3] etc. expressions will give error. To access the value associated with a key, we can mention the key name inside the square braces, as: dict[“Name”].
If we want to know how many key-value pairs are there in a dictionary, we can use the len( ) function, as shown

```
d= {“Regd.No”: 556, “Name”:"Mothi”, “Branch”:
“CSE”}
print len(d)                # 3
```

We can also insert a new key-value pair into an existing dictionary. This is done by mentioning the key and assigning a value to it.

```
d={'Regd.No':556,'Name':'Mothi','Branch':'CSE'}
print d    #{'Branch': 'CSE', 'Name': 'Mothi', 'Regd.No':
556} d['Gender']=’Male’
print d    # {'Gender': 'Male', 'Branch': 'CSE', 'Name': 'Mothi', 'Regd.No': 556}
```

Suppose, we want to delete a key-value pair from the dictionary, we can use *del* statement as:

```
del dict[‘Regd.No’] #{'Gender': 'Male', 'Branch': 'CSE', 'Name': 'Mothi'}
```

To Test whether a “key” is available in a dictionary or not, we can use “in” and “not in” operators. These operators return either True or False.

```
“Name” in d    # check if “Name” is a key in d and returns True / False
```

We can use any data types for value. For example, a value can be a number, string, list, tuple or another dictionary. But keys should obey the rules:

- Keys should be unique. It means, duplicate keys are not allowed. If we enter same key again, the old key will be overwritten and only the new key will be available.

```
emp={'nag':10,'vishnu':20,'nag':20}
print emp # {'nag': 20, 'vishnu': 20}
```

- Keys should be immutable type. For example, we can use a number, string or tuples as keys since they are immutable. We cannot use lists or dictionaries as keys. If they are used as keys, we will get “TypeError”.

```
emp={['nag']:10,'vishnu':20,'nag':20}
Traceback (most recent call last):
    File "<pyshell#2>", line 1, in <module>
      emp={['nag']:10,'vishnu':20,'nag':20}
TypeError: unhashable type: 'list'
```

Using for loop with Dictionaries:

for loop is very convenient to retrieve the elements of a dictionary. Let’s take a simple dictionary that contains color code and its name as:

```
colors = { 'r':"RED", 'g':"GREEN", 'b':"BLUE", 'w':"WHITE" }
```

Here, “r”, “g”, “b”, “w” represents keys and “RED”, “GREEN”, “BLUE” and “WHITE” indicate values.

```
colors = { 'r':"RED", 'g':"GREEN", 'b':"BLUE", 'w':"WHITE" }
for k in colors:
        print k # displays only keys
for k in colors:
```

```
print colors[k] # keys to to dictionary and display the values
```

Dictionary Methods:

Method	Description
`d.clear()`	Removes all key-value pairs from dictionary "d".
`d2=d.copy()`	Copies all elements from "d" into a new dictionary d2.
`d.fromkeys(s [,v] )`	Create a new dictionary with keys from sequence "s" and values all set to "v".
`d.get(k [,v] )`	Returns the value associated with key "k". If key is not found, it returns "v".
`d.items()`	Returns an object that contains key-value pairs of "d". The pairs are stored as tuples in the object.
`d.keys()`	Returns a sequence of keys from the dictionary "d".
`d.values()`	Returns a sequence of values from the dictionary "d".
`d.update(x)`	Adds all elements from dictionary "x" to "d".
`d.pop(k [,v] )`	Removes the key "k" and its value from "d" and returns value. If key is not found, then the value "v" is returned. If key is not found and "v" is not mentioned then "KeyError" is raised.
`d.setdefault(k [,v] )`	If key "k" is found, its value is returned. If key is not found, then the k, v pair is stored into the dictionary "d".

Converting Lists into Dictionary:

When we have two lists, it is possible to convert them into a dictionary. For example, we have two lists containing names of countries and names of their capital cities.

There are two steps involved to convert the lists into a dictionary. The first step is to create a "zip" class object by passing the two lists to zip( ) function. The zip( ) function is useful to convert the sequences into a zip class object. The second step is to convert the zip object into a dictionary by using dict( ) function.

Example:

```
countries = [ 'USA', 'INDIA', 'GERMANY', 'FRANCE' ]
cities = [ 'Washington', 'New Delhi', 'Berlin', 'Paris' ]
z=zip(countries, cities)
d=dict(z)
print d
```

Output:

{'GERMANY': 'Berlin', 'INDIA': 'New Delhi', 'USA': 'Washington', 'FRANCE': 'Paris'}

Converting Strings into Dictionary:

When a string is given with key and value pairs separated by some delimiter like a comma (,) we can convert the string into a dictionary and use it as dictionary.

```
s="Vijay=23,Ganesh=20,Lakshmi=19,Nikhil=22"
s1=s.split(',')
s2=[]
d={}
for i in s1:
    s2.append(i.split('='))
```

```
print d
```

```
{'Ganesh': '20', 'Lakshmi': '19', 'Nikhil': '22', 'Vijay': '23'}
```

Q) A Python program to create a dictionary and find the sum of values.

```
d={'m1':85,'m3':84,'eng':86,'c':91}
sum=0
for i in d.values():
        sum+=i
print sum  # 346
```

Q) A Python program to create a dictionary with cricket player's names and scores in a match. Also we are retrieving runs by entering the player's name.

```
n=input("Enter How many players? ")
d={}
for i in range(0,n):
   k=input("Enter Player name:
   ") v=input("Enter score: ")
   d[k]=v
print d
name=input("Enter name of player for score: ")
print "The Score is",d[name]
```

```
Enter How many players? 3
Enter Player name: "Sachin"
Enter score: 98
Enter Player name: "Sehwag"
Enter score: 91
Enter Player name: "Dhoni"
Enter score: 95
{'Sehwag': 91, 'Sachin': 98, 'Dhoni': 95}
Enter name of player for score: "Sehwag"
The Score is 91
```

SEQUENCES

In Python, Sequences are the general term for ordered sets. Python offers six types of sequences.

Strings

Lists

Tuples

Bytes

BytesArrays

Range()

Strings

A string is a group of characters. Since Python has no provision for arrays, we simply use strings. We can use a pair of single or double quotes. Python is dynamically-typed. Every string object is of the type 'str'.

To declare an empty string, we may use the function str():

```
>>> name=str()
>>> name
>>> type(name)
    <class 'str'>
```

Lists

Since Python does not have arrays, it has lists. A list is an ordered group of items. To declare it, we use square brackets.

```
>>> groceries=['milk','bread','eggs']
>>> groceries[1]
    'bread'
>>> groceries[:2]
    ['milk', 'bread']
```

A Python list can hold all kinds of items; this is what makes it heterogenous.

```
>>> mylist=[1,'2',3.0,False]
```

Also, a list is mutable. This means we can change a value.

```
>>> groceries[0]='cheese'
>>> groceries
    ['cheese', 'bread', 'eggs']
```

Tuples

A tuple, in effect, is an immutable group of items. When we say immutable, we mean we cannot change a single value once we declare it.

```
>>> name=('Ayushi','Sharma')
>>> type(name)
<class'tuple'>
```

We can also use the function tuple().

```
>>> name=tuple(['Ayushi','Sharma'])
```

```
>>> name
('Ayushi','Sharma')
```

A tuple is immutable, let's try changing a value.

```
>>> name[0]='Avery'
Traceback (most recent call last):
File "<pyshell#594>", line 1, in <module>
name[0]='Avery'
TypeError: 'tuple' object does not support item assignment
```

To learn more about tuples, read Tuples in Python.

Also read up on Python namedtuple.

Bytes Sequences

The function bytes() returns an immutable bytes object. Let's take a few examples.

A bytes object is displayed as a sequence of bytes between quotes and preceded by 'b' or 'B':

```
>>> bytes(3) # This initialization produces an empty sequence of 3 bytes.
b'\x00\x00\x00'
>>>
>>> bytes([3]) # Initialized with a list containing 1 member.
b'\x03'
>>>
>>> B'\x00\x00\x00'
b'\x00\x00\x00'
>>>
>>> isinstance(b'\x00\x00\x00', bytes)
True
>>> len(b'\x00\x00\x00')
3 # bytes
>>>
```

The representation '\x00' is not read literally. This representation means a byte with value 0x00.

If a member of a bytes object can be displayed as a printable ASCII character, then it is so displayed..

```
>>> B"123ABC"
b'123ABC'
>>>
>>> B'"\x03 1 2 x y \xE7"'
b'\x03 1 2 x y \xe7'
>>>
>>> b"""\x41\x42\x43"""
b'ABC'
>>> bytes(5)
b'\x00\x00\x00\x00\x00'
>>> bytes([1,2,3,4,5])
b'\x01\x02\x03\x04\x05'
>>> bytes('hello','utf-8')
b 'hello'
```

Here, utf-8 is the encoding we used.

Since it is immutable, if we try to change an item, it will raise a TypeError.

```
>>> a=bytes([1,2,3,4,5])
>>> a
b'\x01\x02\x03\x04\x05' b"""\x41\x42\x43"""
>>> a[4]=3
Traceback (most recent call last):
File "<pyshell#46>", line 1, in <module>
a[4]=3
TypeError: 'bytes' object does not support item assignment
```

Bytes Arrays

A bytesarray object is like a bytes object, but it is mutable. It returns an array of the given byte size.

The bytearray is a mutable sequence of bytes, similar to the bytes object in that each member of the bytearray fits into one byte, and similar to a list in that the bytearray or any slice of it may be changed dynamically.

The bytearray is displayed as a bytes object within parentheses prepended by the word bytearray:

```
>>> a=bytearray(4)
>>> a
bytearray(b'\x00\x00\x00\x00')
>>> a=bytearray(4)
>>> a
bytearray(b'\x00\x00\x00\x00\x01')
>>> a[0]=1
>>> a
bytearray(b'\x01\x00\x00\x00\x01')
>>> a[0]
1
```

If it is done on a list.

```
>>> bytearray([1,2,3,4])
bytearray(b'\x01\x02\x03\x04')
```

Finally, let's try changing a value.

```
>>> a=bytearray([1,2,3,4,5])
>>> a
bytearray(b'\x01\x02\x03\x04\x05')
>>> a[4]=3
>>> a
bytearray(b'\x01\x02\x03\x04\x03')
```

Any bytes object may be converted to a bytearray:

```
>>> ba1 = bytearray(b'\x54\x68\x65 \161\165\151\143\153, brown \x2E\x2e\056\056')
>>> ba1; len(ba1) ; isinstance(ba1, bytearray)
bytearray(b'The quick, brown ....')
21
True
```

f. range():

A range() object lends us a range to iterate on; it gives us a list of numbers.

```
>>> a=range(4)
```

```
>>> type(a)
<class ‘range’>
>>> for i in range(7,0,-1):
print(i)
7
6
5
4
3
2
1
```

We took an entire post on Range in Python.

Python Sequence Operations

Sequences are containers with items accessible by indexing or slicing. Python supports a variety of operations that can be applied to sequence types, including strings, lists, and tuples. The operations on strings are

Concatenation

Concatenation adds the second operand after the first one.

```
>>> 'Ayu'+'shi'
‘Ayushi’
```

Integer Multiplication

We can make a string print twice by multiplying it by 2.

```
>>> 'ba'+'na'*2
‘banana’
```

Membership

To check if a value is a member of a sequence, we use the ‘in’ operator.

```
>>> 'men' in 'Disappointment'
True
```

You can read more about these operations in Python Operators.

Python Slice

Sometimes, we only want a part of a sequence, and not all of it. We do it with the slicing operator.

```
>>> 'Ayushi'[1:4]
```

'yus'

Python Sequence Functions

The functions supported by Python in Sequence are

len()

A very common and useful function to pass a sequence to is len(). It returns the length of the Python sequence.

```
>>> len('Ayushi')
6
```

min() and max()

min() and max() return the lowest and highest values, respectively, in a Python sequence.

```
>>> min('cat')
'a'
>>> max('cat')
't'
```

This comparison is based on ASCII values.

Python Sequence Methods

There are some methods that we can call on a Python sequence:

Python index()

This method returns the index of the first occurrence of a value.

```
>>> 'banana'.index('n')
2
```

Python count()

count() returns the number of occurrences of a value in a Python sequence.

```
>>> 'banana'.count('na')
2
>>> 'banana'.count('a')
```

COMPREHENSIONS

Comprehensions in Python provide us with a short and concise way to construct new sequences (such as lists, set, dictionary etc.) using sequences which have been already defined. Python supports the following 4 types of comprehensions:

List Comprehensions

Dictionary Comprehensions

Set Comprehensions

Generator Comprehensions

List Comprehensions:

List Comprehensions provide an elegant way to create new lists. The following is the basic structure of a list comprehension:

output_list = [*output_exp* for *var* in *input_list* if (*var* satisfies this condition)]

Note that list comprehension may or may not contain an if condition. List comprehensions can contain multiple for (nested list comprehensions).

Example #1:

Suppose we want to create an output list which contains only the even numbers which are present in the input list. Let's see how to do this using for loops and list comprehension and decide which method suits better.

```
# Constructing output list WITHOUT using List comprehension
input_list = [1, 2, 3, 4, 4, 5, 6, 7, 7]
output_list = []
# Using loop for constructing output list
for var in input_list:
    if var % 2 == 0:
        output_list.append(var)
        print("Output List using for loop:", output_list)
```

Output:

Output List using for loop: [2, 4, 4, 6]

```
# Using List comprehensions for constructing output list
input_list = [1, 2, 3, 4, 4, 5, 6, 7, 7]
list_using_comp = [var for var in input_list if var % 2 == 0]
print("Output List using list comprehensions:",list_using_comp)
```

Output:

Output List using list comprehensions: [2, 4, 4, 6]

Example #2:

Suppose we want to create an output list which contains squares of all the numbers from 1 to 9. Let's see how to do this using for loops and list comprehension

```
# Constructing output list using for loop
output_list = []
for var in range(1, 10):
    output_list.append(var ** 2)
    print("Output List using for loop:", output_list)
```

Output

Output List using for loop: [1, 4, 9, 16, 25, 36, 49, 64, 81]

```
# Constructing output list using list comprehension
list_using_comp = [var**2 for var in range(1, 10)]
print("Output List using list comprehension:",  list_using_comp)
```

Output:

Output List using list comprehension: [1, 4, 9, 16, 25, 36, 49, 64, 81]

Dictionary Comprehensions:

Extending the idea of list comprehensions, we can also create a dictionary using dictionary comprehensions. The basic structure of a dictionary comprehension looks like below.

output_dict = {key:value for (key, value) in *iterable* if (key, value satisfy this condition)}

Example #1:

Suppose we want to create an output dictionary which contains only the odd numbers that are present in the input list as keys and their cubes as values. Let's see how to do this using for loops and dictionary comprehension.

```
input_list = [1, 2, 3, 4, 5, 6, 7]
output_dict = {}
# Using loop for constructing output dictionary
for var in input_list:
```

```
    if var % 2 != 0:
        output_dict[var] = var**3
print("Output Dictionary using for loop:",  output_dict )
```

Output:

```
Output Dictionary using for loop: {1: 1, 3: 27, 5: 125, 7: 343}
```

```
# Using Dictionary comprehensions for constructing output dictionary
input_list = [1,2,3,4,5,6,7]
dict_using_comp = {var:var ** 3 for var in input_list if var % 2 != 0}
print("Output Dictionary using dictionary comprehensions:", dict_using_comp)
```

Output:

```
Output Dictionary using dictionary comprehensions: {1: 1, 3: 27, 5: 125, 7: 343}
```

Example #2:

Given two lists containing the names of states and their corresponding capitals, construct a dictionary which maps the states with their respective capitals. Let's see how to do this using for loops and dictionary comprehension.

```
state = ['Gujarat', 'Maharashtra', 'Rajasthan']
capital = ['Gandhinagar', 'Mumbai', 'Jaipur']
output_dict = {}
# Using loop for constructing output dictionary
for (key, value) in zip(state, capital):
    output_dict[key] = value
print("Output Dictionary using for loop:",  output_dict)
```

Output:

```
Output Dictionary using for loop: {'Gujarat': 'Gandhinagar',
                                   'Maharashtra': 'Mumbai',
                                   'Rajasthan': 'Jaipur'}
```

```
# Using Dictionary comprehensions for constructing output dictionary
    state = ['Gujarat', 'Maharashtra', 'Rajasthan']
    capital = ['Gandhinagar', 'Mumbai', 'Jaipur']
    dict_using_comp = {key:value for (key, value) in zip(state, capital)}
    print("Output Dictionary using dictionary comprehensions:",   dict_using_comp)
```

Output:

```
Output Dictionary using dictionary comprehensions: {'Rajasthan': 'Jaipur',
                                                    'Maharashtra': 'Mumbai',
                                                    'Gujarat': 'Gandhinagar'}
```

Set Comprehensions:

Set comprehensions are pretty similar to list comprehensions. The only difference between them is that set comprehensions use curly brackets { }. Let's look at the following example to understand set comprehensions.

Example #1 :

Suppose we want to create an output set which contains only the even numbers that are present in the input list. Note that set will discard all the duplicate values. Let's see how we can do this using for loops and set comprehension.

```
input_list = [1, 2, 3, 4, 4, 5, 6, 6, 6, 7, 7]
output_set = set()
# Using loop for constructing output set
for var in input_list:
    if var % 2 == 0:
        output_set.add(var)
        print("Output Set using for loop:", output_set)
```

Output:

```
Output Set using for loop: {2, 4, 6}
```

```
# Using Set comprehensions for constructing output set
input_list = [1, 2, 3, 4, 4, 5, 6, 6, 6, 7, 7]
set_using_comp = {var for var in input_list if var % 2 == 0}
print("Output Set using set comprehensions:",  set_using_comp)
```

Output:

Output Set using set comprehensions: {2, 4, 6}

Generator Comprehensions:

Generator Comprehensions are very similar to list comprehensions. One difference between them is that generator comprehensions use circular brackets whereas list comprehensions use square brackets. The major difference between them is that generators don't allocate memory for the whole list. Instead, they generate each value one by one which is why they are memory efficient. Let's look at the following example to understand generator comprehension:

Example #1 :

```
input_list = [1, 2, 3, 4, 4, 5, 6, 7, 7]
output_gen = (var for var in input_list if var % 2 == 0)
print("Output values using generator comprehensions:", end = ' ')
for var in output_gen:
    print(var, end = ' ')
```

Output:

Output values using generator comprehensions: 2 4 4 6