MODULE – 3

LISTS AND DICTIONARIES

3.1 LISTS

A list is an ordered sequence of values. It is a data structure in Python. The values inside the lists can be of any type (like integer, float, strings, lists, tuples, dictionaries etc) and are called *elements* or *items.* The elements of the lists are enclosed within square brackets. For example,

```
ls1=[10,-4, 25, 13]
ls2=[“Tiger”, “Lion”, “Cheetah”]
```

Here, ls1 is a list containing four integers, and ls2 is a list containing three strings. A list need not contain data of the same type. We can have mixed types of elements in the list. For example,

```
ls3=[3.5, ‘Tiger’, 10, [3,4]]
```

Here, ls3 contains a float, a string, an integer, and a list. This illustrates that a list can be nested as well.

An empty list can be created in any of the following ways –

```
>>> ls =[]
>>> type(ls)
<class 'list'>
```

or

```
>>> ls =list()
>>> type(ls)
<class 'list'>
```

In fact, list() is the name of a method (a special type of method called a constructor – which will be discussed in Module 4) of the class *list.* Hence, a new list can be created using this function by passing arguments to it as shown below –

```
>>> ls2=list([3,4,1])
>>> print(ls2)
[3, 4, 1]
```

3.1.1 List Operations

Python allows using of operators + and * on lists. The operator + uses two list objects and returns the concatenation of those two lists. Whereas * operator takes one list object and one integer value, say n and returns a list by repeating itself for n times.

```
>>> ls1=[1,2,3]
>>> ls2=[5,6,7]
>>> print(ls1+ls2)   #concatenation using +
[1, 2, 3, 5, 6, 7]

>>> ls1=[1,2,3]
>>> print(ls1*3)     #repetition using *
[1, 2, 3, 1, 2, 3, 1, 2, 3]

>>> [0]*4            #repetition using *
[0, 0, 0, 0]
```

3.1.2 Traversing a List

A list can be traversed using *for* loop. If we need to use each element in the list, we can usethe *for* loop and *in* operator as below –

```
>>> ls=[34, 'hi', [2,3],-5]
>>> for item in ls:
        print(item)

    34
    hi
    Hello
    -5
```

List elements can be accessed with the combination of *range()* and *len()* functions as well –

```
ls=[1,2,3,4]
for i in range(len(ls)):
  ls[i]=ls[i]**2
  print(ls)               #output is
[1, 4, 9, 16]
```

Here, we wanted to do modifications in the elements of the list. Hence, referring to indices is more suitable than referring to elements directly. The *len()* returns a total number of elements in the list (here it is 4). Then *range()* function makes the loop to range from 0 to 3 (i.e. 4-1). Then, for every index, we are updating the list elements (replacing the original value with its square).

3.1.3 List Slices

Similar to strings, the slicing can be applied on lists as well. Consider a list t given below, and a series of examples following based on this object.

```
t=['a','b','c','d','e']
```

- Extracting full list without using any index, but only a slicing operator –

```
>>> print(t[:])
    ['a', 'b', 'c', 'd', 'e']
```

- Extracting elements from 2nd position –

```
>>> print(t[1:])
    ['b', 'c', 'd', 'e']
```

- Extracting first three elements –

```
>>> print(t[:3])
    ['a', 'b', 'c']
```

- Selecting some middle elements –

```
>>> print(t[2:4])
    ['c', 'd']
```

- Using negative indexing –

```
>>> print(t[:-2])
    ['a', 'b', 'c']
```

- **Reversing a list** using negative value for stride –

```
>>> print(t[::-1])
    ['e', 'd', 'c', 'b', 'a']
```

- Modifying (reassignment) only required set of values –

```
>>> t[1:3]=['p','q']
>>> print(t)
    ['a', 'p', 'q', 'd', 'e']
```

Thus, slicing can make many tasks simple.

3.1.4 List Methods

There are several built-in methods in *list* class for various purposes. Here, we will discusssome of them.

- **append():** This method is used to add a new element at the end of a list.

```
>>> ls=[1,2,3]
>>> ls.append('hi')
>>> ls.append(10)
>>> print(ls)
```

```
[1, 2, 3, 'hi', 10]
```

- **extend():** This method takes a list as an argument and all the elements in this listare added at the end of invoking list.

```
>>> ls1=[1,2,3]
>>> ls2=[5,6]
>>> ls2.extend(ls1)
>>> print(ls2)
     [5, 6, 1, 2, 3]
```

Now, in the above example, the list ls1 is unaltered.

- **sort():** This method is used to sort the contents of the list. By default, the functionwill sort the items in ascending order.

```
>>> ls=[3,10,5, 16,-2]
>>> ls.sort()
>>> print(ls)
     [-2, 3, 5, 10, 16]
```

When we want a list to be sorted in descending order, we need to set the argumentas shown –

```
>>> ls.sort(reverse=True)
>>> print(ls)
[16, 10, 5, 3, -2]
```

- **reverse():** This method can be used to reverse the given list.

```
>>> ls=[4,3,1,6]
>>> ls.reverse()
>>> print(ls)
     [6, 1, 3, 4]
```

- **count():** This method is used to count number of occurrences of a particular valuewithin list.

```
>>> ls=[1,2,5,2,1,3,2,10]
>>> ls.count(2)
     3               #the item 2 has appeared 3 tiles
     in ls
```

- **clear():** This method removes all the elements in the list and makes the list empty.

```
>>> ls=[1,2,3]
>>> ls.clear()
>>> print(ls)
      []
```

- **insert():** Used to insert a value before a specified index of the list.

```
>>> ls=[3,5,10]
>>> ls.insert(1,"hi")
```

```
>>> print(ls)
     [3, 'hi', 5, 10]
```

- **index():** This method is used to get the index position of a particular value in the list.

```
>>> ls=[4, 2, 10, 5, 3, 2, 6]
>>> ls.index(2)
     1
```

Here, the number 2 is found at the index position 1. Note that, this function will give index of only the first occurrence of a specified value. The same function can beused with two more arguments *start* and *end* to specify a range within which the search should take place.

```
>>> ls=[15, 4, 2, 10, 5, 3, 2, 6]
>>> ls.index(2)
     2
>>> ls.index(2,3,7)
    6
```

If the value is not present in the list, it throws ValueError.

```
>>> ls=[15, 4, 2, 10, 5, 3, 2, 6]
>>> ls.index(53)
     ValueError: 53 is not in list
```

Few important points about List Methods:

1. There is a difference between *append()* and *extend()* methods. The former adds the argument as it is, whereas the latter enhances the existing list. To understand this, observe the following example –

```
>>> ls1=[1,2,3]
>>> ls2=[5,6]
>>> ls2.append(ls1)
>>> print(ls2)
   [5, 6, [1, 2, 3]]
```

Here, the argument ls1 for the *append()* function is treated as one item, and made as an inner list to ls2. On the other hand, if we replace *append()* by *extend()* then the result would be –

```
>>> ls1=[1,2,3]
>>> ls2=[5,6]
>>> ls2.extend(ls1)
>>>  print(ls2)
[5, 6, 1, 2, 3]
```

2. The ***sort()*** function can be applied only when the list contains elements of compatible types. But, if a list is a mix non-compatible

types like integers and string, the comparisoncannot be done. Hence, Python will throw TypeError. For example,

```
>>> ls=[34, 'hi', -5]
>>> ls.sort()
TypeError: '<' not supported between instances of 'str' and
'int'
```

Similarly, when a list contains integers and sub-list, it will be an error.

```
>>> ls=[34,[2,3],5]
>>> ls.sort()
TypeError: '<' not supported between instances of 'list'
and 'int'
```

Integers and floats are compatible and relational operations can be performed on them.Hence, we can sort a list containing such items.

```
>>> ls=[3, 4.5, 2]
>>> ls.sort()
>>> print(ls)
     [2, 3, 4.5]
```

3. The ***sort()*** function uses one important argument ***keys***. When a list is containing tuples, it will be useful. We will discuss tuples later in this Module.

4. Most of the list methods like *append()*, *extend()*, *sort()*, *reverse()* etc. modify the list object internally and return None.

```
>>> ls=[2,3]
>>> ls1=ls.append(5)
>>> print(ls)
     [2,3,5]
>>>
    print (ls1)
None
```

3.1.5 Deleting Elements

Elements can be deleted from a list in different ways. Python provides few built-in methodsfor removing elements as given below –

- **pop():** This method deletes the last element in the list, by default.

```
>>> ls=[3,6,-2,8,10]
>>> x=ls.pop() #10 is removed from list and stored in x
>>> print(ls)
[3, 6, -2, 8]
>>>print(x)
10
```

When an element at a particular index position has to be deleted, then we can givethat position as argument to *pop()* function.

```
>>> t = ['a', 'b', 'c']
>>> x = t.pop(1)          #item at index 1 is popped
>>> print(t)
    ['a', 'c']
>>> print(x)
    b
```

- **remove():** When we don't know the index, but know the value to be removed, thenthis function can be used.

```
>>> ls=[5,8, -12,34,2]
>>> ls.remove(34)
>>> print(ls)
     [5, 8, -12, 2]
```

Note that, this function will remove only the first occurrence of the specified value,but not all occurrences.

```
>>> ls=[5,8, -12, 34, 2, 6, 34]
>>> ls.remove(34)
>>> print(ls)
   [5, 8, -12, 2, 6, 34]
```

Unlike *pop()* function, the *remove()* function will not return the value that has beendeleted.

- **del:** This is an operator to be used when more than one item to be deleted at a time.Here also, we will not get the items deleted.

```
>>> ls=[3,6,-2,8,1]
>>> del ls[2]             #item at index 2 is deleted
>>> print(ls)
   [3, 6, 8, 1]

>>> ls=[3,6,-2,8,1]
>>> del ls[1:4] #deleting all elements from index 1 to 3
>>> print(ls)
         [3, 1]
```

Deleting all odd indexed elements of a list –

```
>>> t=['a', 'b', 'c', 'd', 'e']
>>> del t[1::2]
>>> print(t)
     ['a', 'c', 'e']
```

3.1.6 Lists are Mutable

The elements in the list can be accessed using a numeric index within square-brackets. It issimilar to extracting characters in a string.

```
>>> ls=[34, 'hi', [2,3],-5]
>>> print(ls[1])
     hi
>>> print(ls[2])
     [2, 3]
```

Observe here that, the inner list is treated as a single element by outer list. If we would like to access the elements within inner list, we need to use double-indexing as shown below –

```
>>> print(ls[2][0])
     2
>>> print(ls[2][1])
     3
```

Note that, the indexing for inner-list again starts from 0. Thus, when we are using double- indexing, the first index indicates position of inner list inside outer list, and the second index means the position particular value within inner list.

Unlike strings, lists are mutable. That is, using indexing, we can modify any value within list.In the following example, the 3rd element (i.e. index is 2) is being modified –

```
>>> ls=[34, 'hi', [2,3],-5]
>>> ls[2]='Hello'
>>> print(ls)
     [34, 'hi', 'Hello', -5]
```

The list can be thought of as a relationship between indices and elements. This relationshipis called as a ***mapping***. That is, each index maps to one of the elements in a list.

The index for extracting list elements has following properties –

- Any integer expression can be an index.

```
>>> ls=[34, 'hi', [2,3],-5]
>>> print(ls[2*1])
     'Hello'
```

- Attempt to access a non-existing index will throw and IndexError.

```
>>> ls=[34, 'hi', [2,3],-5]
>>> print(ls[4])
IndexError: list index out of range
```

- A negative indexing counts from backwards.

```
>>> ls=[34, 'hi', [2,3],-5]
>>> print(ls[-1])
      -5
>>> print(ls[-3])
      hi
```

The ***in*** operator applied on lists will results in a Boolean value.

```
>>> ls=[34, 'hi', [2,3],-5]
>>> 34 in ls
      True
>>> -2 in ls
      False
```

3.1.7 Lists and Functions

The utility functions like ***max(), min(), sum(), len()*** etc. can be used on lists. Hence most ofthe operations will be easy without the usage of loops.

```
>>> ls=[3,12,5,26, 32,1,4]
>>> max(ls)             # prints  32
>>> min(ls)             # prints  1
>>> sum(ls)             # prints  83
>>> len(ls)             # prints  7

>>> avg=sum(ls)/len(ls)
>>> print(avg)
   11.857142857142858
```

When we need to read the data from the user and to compute sum and average of those numbers, we can write the code as below –

```
ls= list()
while (True):
   x= input('Enter a number: ')
   if x== 'done':
         break

   x= float(x)
   ls.append(x)

average = sum(ls) / len(ls)
print('Average:', average)
```

In the above program, we initially create an empty list. Then, we are taking an infinite *while*-loop. As every input from the keyboard will be in the form of a string, we need to convert x into float type and then append it to a list. When the keyboard input is a string 'done', then the loop is going to get terminated. After the loop, we will find the average of those numbers with the help of built-in functions *sum()* and *len()*.

3.1.8 Lists and Strings

Though both lists and strings are sequences, they are not same. In fact, a list of characters is not same as string. To convert a string into a list, we use a method ***list()*** as below –

```
>>> s="hello"
>>> ls=list(s)
>>> print(ls)
     ['h', 'e', 'l', 'l', 'o']
```

The method ***list()*** breaks a string into individual letters and constructs a list. If we want a listof words from a sentence, we can use the following code –

```
>>> s="Hello how are you?"
>>> ls=s.split()
>>> print(ls)
     ['Hello', 'how', 'are', 'you?']
```

Note that, when no argument is provided, the ***split()*** function takes the delimiter as white space. If we need a specific delimiter for splitting the lines, we can use as shown in following example –

```
>>> dt="20/03/2018"
>>> ls=dt.split('/')
>>> print(ls)
     ['20', '03', '2018']
```

There is a method ***join()*** which behaves opposite to ***split()*** function. It takes a list of strings as argument, and joins all the strings into a single string based on the delimiter provided. For example –

```
>>> ls=["Hello", "how", "are", "you"]
>>> d=' '
>>> d.join(ls)
'Hello how are you'
```

Here, we have taken delimiter d as white space. Apart from space, anything can be taken as delimiter. When we don't need any delimiter, use empty string as delimiter.

3.1.9 Parsing Lines

In many situations, we would like to read a file and extract only the lines containing requiredpattern. This is known as ***parsing***. As an illustration, let us assume that there is a log file containing details of email communication between employees of an organization. For all received mails, the file contains lines as –

From ManojkumarSB@bgsit.ac.in *Sat Jul 03 09:14:16 2023*

From sbmanojkumar@bgsit.ac.in *Sun Jul 4 06:12:51 2023*

....

Apart from such lines, the log file also contains mail-contents, to-whom the mail has been sent etc. Now, if we are interested in extracting only the days of incoming mails, then we can go for parsing. That is, we are interested in knowing on which of the days, the mails have been received. The code would be –

```
fhand = open('logFile.txt')
for line in fhand:
     line = line.rstrip()
     if not line.startswith('From '):
          continue
     words =
     line.split()
     print(words[2])
```

Obviously, all received mails starts from the word From. Hence, we search for only such lines and then split them into words. Observe that, the first word in the line would be From, second word would be email-ID and the 3rd word would be day of a week. Hence, we will extract words[2] which is 3rd word.

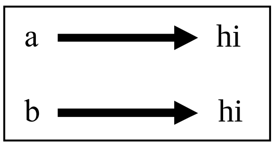
3.1.10 Objects and Values

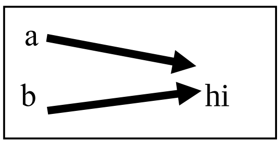
Whenever we assign two variables with same value, the question arises – whether both thevariables are referring to same object, or to different objects. This is important aspect to know, because in Python everything is a class object. There is nothing like elementary datatype.

Consider a situation –

a= "hi"
b= "hi"

Now, the question is whether both a and b refer to the ***same string***. There are two possible states –





In the first situation, a and b are two different objects, but containing same value. The modification in one object is nothing to do with the other. Whereas, in the second case,both a and b are referring to the same object. That is, a is an ***alias name*** for b and vice- versa. In other words, these two are referring to same memory location.

To check whether two variables are referring to same object or not, we can use ***is*** operator.

```
>>> a= "hi"
>>> b= "hi"
>>> a is b          #result is True
>>> a==b            #result is True
```

When two variables are referring to same object, they are called as ***identical objects.*** When two variables are referring to different objects, but contain a same value, they are known as ***equivalent objects***. For example,

```
>>> s1=input("Enter a string:")#assume you entered hello
>>> s2=input("Enter a string:")#assume you entered hello

>>> s1 is s2         #check s1 and s2 are identical
False
>>> s1 == s2         #check s1 and s2 are equivalent

True
```

Here **s1** and **s2** are equivalent, but not identical.

If two objects are identical, they are also equivalent, but if they are equivalent, they are not necessarily identical.

String literals are ***interned*** by default. That is, when two string literals are created in the program with the same value, they are going to refer the same object. But, string variables read from the key-board will not have this behavior, because their values are depending on the user's choice.

Lists are not interned. Hence, we can see following result –

```
>>> ls1=[1,2,3]
>>> ls2=[1,2,3]
>>> ls1 is ls2      #output is False
>>> ls1 == ls2      #output is True
```

3.1.11 Aliasing

When an object is assigned to other using the assignment operator, both of them will refer to the same object in the memory. The association of a variable with an object is called a ***reference.***

```
>>> ls1=[1,2,3]
>>> ls2= ls1
```

3.1.12 List Arguments

When a list is passed to a function as an argument, then function receives a reference to this list. Hence, if the list is modified within a function, the caller will get the modified version. Consider an example –

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

ls = ['a', 'b', 'c']
del_front(ls)
print(ls)       # output is ['b', 'c']
```

Now, ls2 is said to be ***reference*** of ls1. In other words, there are two references to thesame object in the memory.

An object with more than one reference has more than one name, hence we say that the object is ***aliased.*** If the aliased object is mutable, changes made in one alias will reflect the other.

```
>>> ls2[1]= 34
>>> print(ls1)       #output is [1, 34, 3]
```

Strings are safe in this regard, as they are immutable.

Here, the argument lsand the parameter tboth are aliases to same object.

One should understand the operations that will modify the list and the operations that create a new list. For example, the ***append()*** function modifies the list, whereas the + operator creates a new list.

```
>>> t1 = [1, 2]
>>> t2 = t1.append(3)
>>> print(t1)               #output is [1 2 3]
>>> print(t2)               #prints None

>>> t3 = t1 + [5]
>>> print(t3)               #output is [1 2 3 5]
>>> t2 is t3                #output is False
```

Here, after applying *append()* on t1 object, the t1 itself has been modified and t2 is notgoing to get anything. But, when the + operator is applied, t1 remains the same but t3 will get the updated result.

The programmer should understand such differences when he/she creates a functionintending to modify a list. For example, the following function has no effect on the original list –

```
def test(t):
    t=t[1:]

ls=[1,2,3]
test(ls)
```

```
print(ls)      #prints [1, 2, 3]
```

One can write a return statement after slicing as below –

```
def test(t):
     return t[1:]

ls=[1,2,3]
ls1=test(ls)
print(ls1)     #prints [2, 3]
print(ls)      #prints [1, 2, 3]
```

In the above example also, the original list is not modified, because a return statement always creates a new object and is assigned to LHS variable at the position of function call.

3.2 DICTIONARIES

A dictionary is a collection of unordered set of ***key:value*** pairs, with the requirement that keys are unique in one dictionary. Unlike lists and strings where elements are accessed using index values (which are integers), the values in dictionary are accessed using keys. A key in dictionary can be any immutable type like strings, numbers and tuples. (The tuple can be made as a key for dictionary, only if that tuple consist of string/number/ sub-tuples). As lists are mutable – that is, can be modified using index assignments, slicing, or using methods like *append()*, *extend()* etc, they cannot be a key for dictionary.

One can think of a dictionary as a mapping between set of indices (which are actually keys)and a set of values. Each key maps to a value.

An empty dictionary can be created using two ways –

```
d= {}
```

OR

```
d=dict()
```

To add items to dictionary, we can use square brackets as –

```
>>> d={}
>>> d["Mango"]="Fruit"
>>> d["Banana"]="Fruit"
>>> d["Cucumber"]="Veg"
>>> print(d)
{'Mango': 'Fruit', 'Banana': 'Fruit', 'Cucumber': 'Veg'}
```

To initialize a dictionary at the time of creation itself, one can use the code like –

```
>>> tel_dir={'Tom': 3491, 'Jerry':8135}
>>> print(tel_dir)
     {'Tom': 3491, 'Jerry': 8135}

>>> tel_dir['Donald']=4793
>>> print(tel_dir)
     {'Tom': 3491, 'Jerry': 8135, 'Donald': 4793}
```

NOTE that the order of elements in dictionary is unpredictable. That is, in the above example, don't assume that 'Tom': 3491 is first item, 'Jerry': 8135 is second item etc. As dictionary members are not indexed over integers, the order of elements inside it may vary. However, using a *key,* we can extract its associated value as shown below –

```
>>> print(tel_dir['Jerry'])
     8135
```

Here, the key 'Jerry'maps with the value 8135, hence it doesn't matter where exactly itis inside the dictionary.

If a particular key is not there in the dictionary and if we try to access such key, then the *KeyError* is generated.

```
>>> print(tel_dir['Mickey'])
     KeyError: 'Mickey'
```

The ***len()*** function on dictionary object gives the number of key-value pairs in that object.

```
>>> print(tel_dir)
     {'Tom': 3491, 'Jerry': 8135, 'Donald': 4793}
>>> len(tel_dir)
     3
```

The ***in*** operator can be used to check whether any ***key*** (not value) appears in the dictionaryobject.

```
>>> 'Mickey' in tel_dir        #output is False
>>> 'Jerry' in tel_dir         #output is True
>>> 3491 in tel_dir            #output is False
```

We observe from above example that the value 3491 is associated with the key 'Tom'in tel_dir. But, the ***in*** operator returns False.

The dictionary object has a method ***values()*** which will ***return a list*** of all the values associated with keys within a dictionary. If we would like to check whether a particular valueexist in a dictionary, we can make use of it as shown below –

```
>>> 3491 in tel_dir.values()  #output is True
```

The ***in*** operator behaves differently in case of lists and dictionaries as explained hereunder–

- When ***in*** operator is used to search a value in a list, then *linear search* algorithm is used internally. That is, each element in the list is checked one by one sequentially. This is considered to be expensive in the view of total time taken to process. Because, if there are 1000 items in the list, and if the element in the list which we aresearch for is in the last position (or if it does not exists), then before yielding result of search (True or False), we would have done 1000 comparisons. In other words, linear search requires n number of comparisons for the input size of n elements. Time complexity of the linear search algorithm is O(n).

- The keys in dictionaries of Python are basically ***hashable*** elements. The concept of ***hashing*** is applied to store (or maintain) the keys of dictionaries. Normally hashing techniques have the time complexity as *O(log n)* for basic operations like insertion, deletion and searching. Hence, the ***in*** operator applied on keys of dictionaries works better compared to that on lists. (Hashing technique is explained at the end of this Section, for curious readers)

3.2.1 Dictionary as a Collection of Counters

Assume that we need to count the frequency of alphabets in a given string. There are different methods to do it –

- Create 26 variables to represent each alphabet. Traverse the given string andincrement the corresponding counter when an alphabet is found.
- Create a list with 26 elements (all are zero in the beginning) representing alphabets. Traverse the given string and increment corresponding indexed position in the list when an alphabet is found.
- Create a dictionary with characters as keys and counters as values. When we find a character for the first time, we add the item to dictionary. Next time onwards, we increment the value of existing item.

Each of the above methods will perform same task, but the logic of implementation will be different. Here, we will see the implementation using dictionary.

```
s=input("Enter a string:")    #read a string
d=dict()                      #create empty dictionary

for ch in s:            #traverse through string
    if ch not in d:     #if new character found
        d[ch]=1         #initialize counter to 1
     else:             #otherwise, increment counter
        d[ch]+=1

print(d)                #display the dictionary
```

The sample output would be –

```
Enter a string: Hello World
{'H': 1, 'e': 1, 'l': 3, 'o': 2, ' ': 1, 'W': 1, 'r': 1, 'd':
1}
```

It can be observed from the output that, a dictionary is created here with characters as keysand frequencies as values. **Note** that, here we have computed ***histogram*** of counters.

Dictionary in Python has a method called as ***get()***, which takes key and a default value as two arguments. If key is found in the dictionary, then the *get()* function returns corresponding value, otherwise it returns default value. For example,

```
>>> tel_dir={'Tom': 3491, 'Jerry':8135, 'Mickey':1253}
>>> print(tel_dir.get('Jerry',0))
     8135
>>> print(tel_dir.get('Donald',0))
     0
```

In the above example, when the *get()* function is taking 'Jerry' as argument, it returned corresponding value, as 'Jerry'is found in tel_dir. Whereas, when *get()* is used with'Donald' as key, the default value 0 (which is provided by us) is returned.

The function ***get()*** can be used effectively for calculating frequency of alphabets in a string. Here is the modified version of the program –

```
s=input("Enter a string:")
d=dict()

for ch in s:
   d[ch]=d.ge
   t(ch,0)+1

print(d)
```

In the above program, for every character ch in a given string, we will try to retrieve a value. When the ch is found in d, its value is retrieved, 1 is added to it, and restored. If ch is not found, 0 is taken as default and then 1 is added to it.

3.2.2 Looping and Dictionaries

When a *for*-loop is applied on dictionaries, it will iterate over the keys of dictionary. If we want to print key and values separately, we need to use the statements as shown –

```
tel_dir={'Tom': 3491, 'Jerry':8135, 'Mickey':1253}
for k in tel_dir:
     print(k, tel_dir[k])
```

Output would be –

```
Tom 3491
Jerry 8135
Mickey 1253
```

Note that, while accessing items from dictionary, the keys may not be in order. If we want toprint the keys in alphabetical order, then we need to make a list of the keys, and then sort that list. We can do so using ***keys()*** method of dictionary and ***sort()*** method of lists. Consider the following code –

```
tel_dir={'Tom': 3491, 'Jerry':8135, 'Mickey':1253}
ls=list(tel_dir.keys())
print("The list of keys:",ls)
ls.sort()
print("Dictionary elements in alphabetical order:")
for k in ls:
    print(k, tel_dir[k])
```

The output would be –

```
The list of keys: ['Tom', 'Jerry', 'Mickey']
Dictionary elements in alphabetical order:
Jerry 8135
Mickey 1253
Tom 3491
```

Note: The key-value pair from dictionary can be together accessed with the help of amethod ***items()*** as shown –

```
>>> d={'Tom':3412, 'Jerry':6781, 'Mickey':1294}
>>> for k,v in d.items():
```

```
                print(k,v)
```

Output:

```
Tom 3412
Jerry 6781
Mickey 1294
```

The usage of comma-separated list k,v here is internally a tuple (another data structure inPython, which will be discussed later).

3.2.3 Reverse lookup

Given a dictionary d and a key k, it is easy to find the corresponding value v = d[k]. This operation is called a **lookup**. But what if you have v and you want to find k? You have two problems: first, there might be more than one key that maps to the value v. Depending on the application, you might be able to pick one, or you might have to make a list that contains all of them. Second, there is no simple syntax to do a **reverse lookup**; you have to search. Here is a function that takes a value and returns the first key that maps to that value:

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

This function is yet another example of the search pattern, but it uses a feature we haven't seen before, raise. The **raise statement** causes an exception; in this case it causes a LookupError, which is a built-in exception used to indicate that a lookup operation failed. If we get to the end of the loop, that means v doesn't appear in the dictionary as a value, so we raise an exception. Here is an example of a successful reverse lookup:

```
>>> h = histogram('parrot')
>>> key = reverse_lookup(h, 2)
>>> key
'r'
And an unsuccessful one:
>>> key = reverse_lookup(h, 3)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
File "<stdin>", line 5, in reverse_lookup
LookupError
```

The effect when you raise an exception is the same as when Python raises one: it prints a traceback and an error message. The raise statement can take a detailed error message as an optional argument. For example:

```
>>> raise LookupError('value does not appear in the
dictionary')
Traceback (most recent call last):
File "<stdin>", line 1, in ?
```

LookupError: value does not appear in the dictionary

A reverse lookup is much slower than a forward lookup; if you have to do it often, or if the dictionary gets big, the performance of your program will suffer.

3.2.4 Dictionaries and lists

Lists can appear as values in a dictionary. For example, if you are given a dictionary that maps from letters to frequencies, you might want to invert it; that is, create a dictionary that maps from frequencies to letters. Since there might be several letters with the same frequency, each value in the inverted dictionary should be a list of letters. Here is a function that inverts a dictionary:

```
def invert_dict(d):
   inverse = dict()
   for key in d:
      val = d[key]
      if val not in inverse:
         inverse[val] = [key]
      else:
         inverse[val].append(key)
   return inverse
```

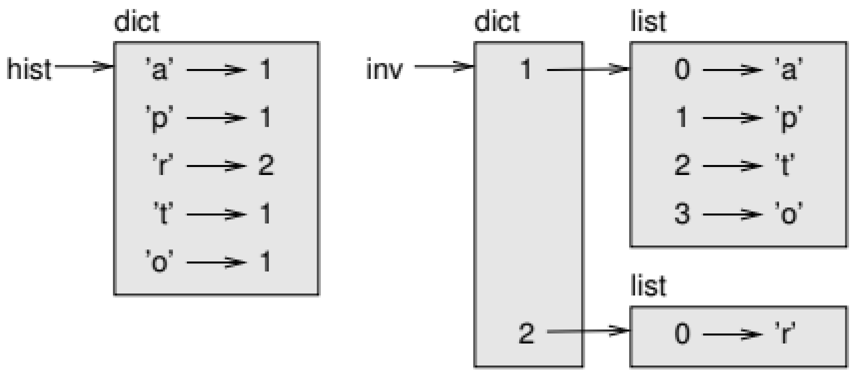


Figure 3.1: State diagram.

Each time through the loop, key gets a key from d and val gets the corresponding value. If val is not in inverse, that means we haven't seen it before, so we create a new item and initialize it with a **singleton** (a list that contains a single element). Otherwise, we have seen this value before, so we append the corresponding key to the list.

Here is an example:

```
>>> hist = histogram('parrot')
>>> hist
{'a': 1, 'p': 1, 'r': 2, 't': 1, 'o': 1}
>>> inverse = invert_dict(hist)
>>> inverse
{1: ['a', 'p', 't', 'o'], 2: ['r']}
```

Figure 3.1 is a state diagram showing hist and inverse. A dictionary is represented as a box with the type dict above it and the key-value pairs inside. If the values are integers, floats or strings, I draw them inside the box, but I usually draw lists outside the box, just to keep the diagram simple. Lists can be values in a dictionary, as this example shows, but they cannot be keys. Here's what happens if you try:

```
>>> t = [1, 2, 3]
>>> d = dict()
>>> d[t] = 'oops'
Traceback (most recent call last):
File "<stdin>", line 1, in ?
TypeError: list objects are unhashable
```

I mentioned earlier that a dictionary is implemented using a hashtable and that means that the keys have to be **hashable**.

A **hash** is a function that takes a value (of any kind) and returns an integer. Dictionaries use these integers, called hash values, to store and look up key-value pairs.

3.2.5 Dictionaries and Files

A dictionary can be used to count the frequency of words in a file. Consider a file *myfile.txt*
consisting of following text –

hello, how are you?
I am doing fine.
How about you?

Now, we need to count the frequency of each of the word in this file. So, we need to takean outer loop for iterating over entire file, and an inner loop for traversing each line in a file. Then in every line, we count the occurrence of a word, as we did before for a character. The program is given as below –

```
fname=input("Enter file name:")
try:
    fhand=ope
n(fname)
except:
    print("File cannot be opened")
    exit()

d=dict()

for line in fhand:
    for word in line.split():
        d[word]=d.get(word,0)+1

print(d)
```

The output of this program when the input file is *myfile.txt* would be –

```
Enter file name: myfile.txt
{'hello,': 1, 'how': 1, 'are': 1, 'you?': 2, 'I': 1,
'am': 1,'doing': 1, 'fine.': 1, 'How': 1, 'about': 1}
```

Few points to be observed in the above output –

- The punctuation marks like comma, full point, question mark etc. are also considered as a part of word and stored in the dictionary. This means, when a particular word appears in a file with and without punctuation mark, then there will bemultiple entries of that word.
- The word 'how' and 'How' are treated as separate words in the above example because of uppercase and lowercase letters.

While solving problems on text analysis, machine learning, data analysis etc. such kinds of treatment of words lead to unexpected results. So, we need to be careful in parsing the textand we should try to eliminate punctuation marks, ignoring the case etc. The procedure is discussed in the next section.

3.2.6 Advanced Text Parsing

As discussed in the previous section, during text parsing, our aim is to eliminate punctuation marks as a part of word. The *string* module of Python provides a list of all punctuation marks as shown –

```
>>> import string
>>> string.punctuation
    '!"#$%&\'()*+,-./:;<=>?@[\\]^_`{|}~'
```

The *str* class has a method *maketrans()* which returns a translation table usable for anothermethod *translate()*. Consider the following syntax to understand it more clearly –

```
line.translate(str.maketrans(fromstr, tostr, deletestr))
```

The above statement replaces the characters in fromstr with the character in the same position in tostr and delete all characters that are in deletestr. The fromstr and tostr can be empty strings and the deletestr parameter can be omitted.

Using these functions, we will re-write the program for finding frequency of words in a file.

```
import string

fname=input("Enter

file name:")try:
    fhand=open(fname)
except:
    print("File cannot be opened")
    exit()

d=dict()
for line in
     fhand:
     line=line.r
     strip()
     line=line.translate(line.maketrans('','',string.punctuati
     on)) line=line.lower()

     for word in
          line.split():
          d[word]=d.get(wor
          d,0)+1

print(d)
```

Now, the output would be –

```
Enter file name:myfile.txt
{'hello': 1, 'how': 2, 'are': 1, 'you': 2, 'i': 1,
'am': 1, 'doing': 1, 'fine': 1, 'about': 1}
```

Comparing the output of this modified program with the previous one, we can make out thatall the punctuation marks are not considered for parsing and also the case of the alphabets are ignored.

3.2.7 Memos

If you played with the fibonacci function from Section 6.7, you might have noticed that the bigger the argument you provide, the longer the function takes to run. Furthermore, the run time increases quickly.

To understand why, consider Figure 3.2, which shows the **call graph** for fibonacci with n=4:

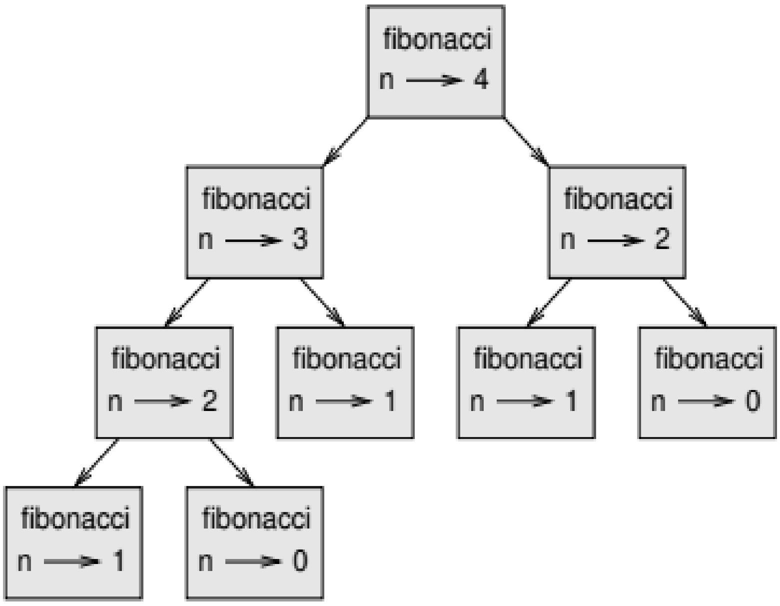


Figure 3.2: Call graph

A call graph shows a set of function frames, with lines connecting each frame to the frames of the functions it calls. At the top of the graph, fibonacci with n=4 calls fibonacci with n=3 and n=2. In turn, fibonacci with n=3 calls fibonacci with n=2 and n=1. And so on. Count how many times fibonacci(0) and fibonacci(1) are called. This is an inefficient solution to the problem, and it gets worse as the argument gets bigger. One solution is to keep track of values that have already been computed by storing them in a dictionary. A previously computed value that is stored for later use is called a **memo**. Here is a “memoized” version of fibonacci:

```
known = {0:0, 1:1}
def fibonacci(n):
if n in known:
return known[n]
res = fibonacci(n-1) + fibonacci(n-2)
known[n] = res
return res
```

known is a dictionary that keeps track of the Fibonacci numbers we already know. It starts with two items: 0 maps to 0 and 1 maps to 1. Whenever fibonacci is called, it checks known. If the result is already there, it can return immediately. Otherwise, it has to compute the new value, add it to the dictionary, and return it. If you run this version of fibonacci and compare it with the original, you will find that it is much faster

3.2.8 Debugging

When we are working with big datasets (like file containing thousands of pages), it is difficult to debug by printing and checking the data by hand. So, we can follow any of the following procedures for easy debugging of the large datasets –

- **Scale down the input**: If possible, reduce the size of the dataset. For example if the program reads a text file, start with just first 10 lines or with the smallest example you can find. You can either edit the files themselves, or modify the program so it reads onlythe first n lines. If there is an error, you can reduce n to the smallest value that manifests the error, and then increase it gradually as you correct the errors.
- **Check summaries and types**: Instead of printing and checking the entire dataset, consider printing summaries of the data: for example, the number of items in a dictionary or the total of a list of numbers. A common cause of runtime errors is a value that is not the right type. For debugging this kind of error, it is often enough to print the type of a value.
- **Write self-checks**: Sometimes you can write code to check for errors automatically. For example, if you are computing the average of a list of numbers, you could check that theresult is not greater than the largest element in the list or less than the smallest. This is called a ***sanity check*** because it detects results that are "completely illogical". Another kind of check compares the results of two different computations to see if they are consistent. This is called a ***consistency check***.
- **Pretty print the output**: Formatting debugging output can make it easier to spot an error.

Hashing Technique (For curious minds – Only for understanding, not forExams!!)

Hashing is a way of representing dictionaries (Not a Python data structure Dictionary!!). Dictionary is an abstract data type with a set of operations searching, insertion and deletiondefined on its elements. The elements of dictionary can be numeric or characters or most of the times, records. Usually, a record consists of several fields; each may be of different data types. For example, student record may contain student id, name, gender, marks etc. Every record is usually identified by some ***key***. Hashing technique is very useful in database management, because it is considered to be very efficient searching technique.

Here we will consider the implementation of a dictionary of *n* records with keys k1, k2 …kn.Hashing is based on the idea of distributing keys among a one-dimensional array

H[0…m-1], called ***hash table***.

For each key, a value is computed using a predefined function called ***hash function***. This function assigns an integer, called ***hash address***, between 0 to m-1 to each key. Based on the hash address, the keys will be distributed in a hash table.

For example, if the keys k1, k2, …., kn are integers, then a hash function can be h(K) = K mod m.

Let us take keys as 65, 78, 22, 30, 47, 89. And let hash function be,
h(k) = k%10.

Then the hash addresses may be any value from 0 to 9. For each key, hash address willbe computed as –
h(65) = 65 %10 = 5
h(78) = 78%10 = 8
h(22)= 22 % 10 =2
h(30)= 30 %10 =0
h(47) = 47 %10 = 7
h(89)=89 % 10 = 9

Now, each of these keys can be hashed into a hash table as –

0	1	2	3	4	5	6	7	8	9
30		22			65		47	78	89

In general, a hash function should satisfy the following requirements:

- A hash function needs to distribute keys among the cells of hash table as evenly aspossible.
- A hash function has to be easy to compute.

Hash Collisions: Let us have *n* keys and the hash table is of size *m* such that *m<n*. As each key will have an address with any value between 0 to m-1, it is obvious that more than one key will have same hash address. That is, two or more keys need to be hashed into the same cell of hash table. This situation is called as ***hash collision***.

In the worst case, all the keys may be hashed into same cell of hash table. But, we can avoid this by choosing proper size of hash table and hash function. Anyway, every hashing scheme must have a mechanism for resolving hash collision. There are two methods for hash collision resolution, viz.

- Open hashing
- closed hashing

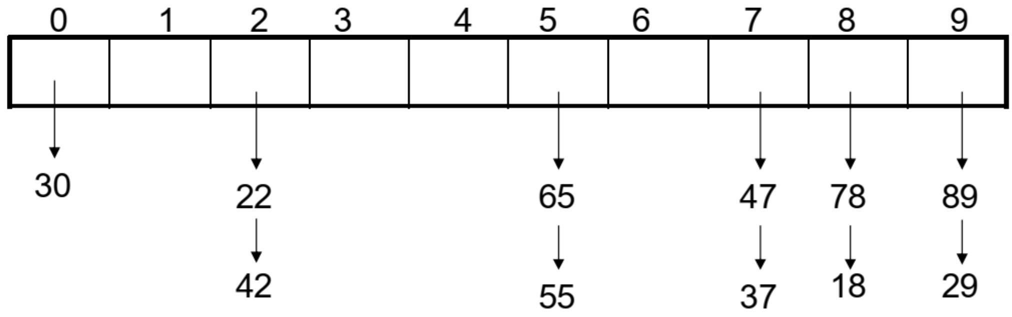
Open Hashing (or Separate Chaining): In open hashing, keys are stored in linked lists attached to cells of a hash table. Each list contains all the keys hashed to its cell. For example, consider the elements

65, 78, 22, 30, 47, 89, 55, 42, 18, 29, 37.

If we take the hash function as h(k)= k%10, then the hash addresses will be

–h(65) = 65 %10 = 5	h(78) = 78%10 = 8
h(22)= 22 % 10 =2	h(30)= 30 %10 =0
h(47) = 47 %10 = 7	h(89)=89 % 10 = 9
h(55)=55%10 =5	h(42)=42%10 =2
h(18)=18%10 =8	h(29)=29%10=9
h(37)=37%10 =7	

The hash table would be –



Operations on Hashing:

- **Searching:** Now, if we want to search for the key element in a hash table, we need to find the hash address of that key using same hash function. Using the obtained hash address, we need to search the linked list by tracing it, till either the key is found or list gets exhausted.
- **Insertion**: Insertion of new element to hash table is also done in

similar manner. Hash key is obtained for new element and is inserted at the end of the list for that particular cell.

- **Deletion**: Deletion of element is done by searching that element and then deleting it from a linked list.

Closed Hashing (or Open Addressing): In this technique, all keys are stored in the hash table itself without using linked lists. Different methods can be used to resolve hash collisions. The simplest technique is ***linear probing.***

This method suggests to check the next cell from where the collision occurs. If that cell is empty, the key is hashed there. Otherwise, we will continue checking for the empty cell in a circular manner. Thus, in this technique, the hash table size must be at least as large as the total number of keys. That is, if we have n elements to be hashed, then the size of hashtable should be greater or equal to n.

Example: Consider the elements 65, 78, 18, 22, 30, 89, 37, 55, 42
Let us take the hash function as h(k)= k%10, then the hash addresses will be –h(65) = 65 %10 = 5 h(78) = 78%10 = 8
h(18)=18%10 =8 h(22)= 22 % 10 =2
h(30)= 30 %10 =0 h(89)=89 % 10 = 9
h(37)=37%10 =7 h(55)=55%10 =5
h(42)=42%10 =2

Since there are 9 elements in the list, our hash table should at least be of size 9. Here weare taking the size as 10.
Now, hashing is done as below –

0	1	2	3	4	5	6	7	8	9
30	89	22	42		65	55	37	78	18

Drawbacks:

- Searching may become like a linear search and hence not efficient.