5.1 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 it is 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 dictionary object.

```
>>> '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 value exist 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 are search 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)

5.1.1 Dictionary as a Set 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 and increment 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 keys and 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.get(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.

5.1.2 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 to print 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 a method ***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 in Python, which will be discussed later).

5.1.3 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 take an 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=open(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 be multiple 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 text and we should try to eliminate punctuation marks, ignoring the case etc. The procedure is discussed in the next section.

5.1.4 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 another method *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.rstrip()
     line=line.translate(line.maketrans('','',string.punctuation))
     line=line.lower()

     for word in line.split():
          d[word]=d.get(word,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 that all the punctuation marks are not considered for parsing and also the case of the alphabets are ignored.

5.2 TUPLES

A tuple is a sequence of items, similar to lists. The values stored in the tuple can be of any type and they are indexed using integers. Unlike lists, tuples are immutable. That is, values within tuples cannot be modified/reassigned. Tuples are *comparable* and *hashable* objects. Hence, they can be made as keys in dictionaries.

A tuple can be created in Python as a comma separated list of items – may or may not be enclosed within parentheses.

```
>>> t='Mango', 'Banana', 'Apple'        #without parentheses
>>> print(t)
     ('Mango', 'Banana', 'Apple')

>>> t1=('Tom', 341, 'Jerry')            #with parentheses
>>> print(t1)
     ('Tom', 341, 'Jerry')
```

Observe that tuple values can be of mixed types.

If we would like to create a tuple with single value, then just a parenthesis will not suffice. For example,

```
>>> x=(3)           #trying to have a tuple with single item
>>> print(x)
     3              #observe, no parenthesis found
>>> type(x)
<class 'int'>       #not a tuple, it is integer!!
```

Thus, to have a tuple with single item, we must include a comma after the item. That is,

```
>>> t=3,            #or use the statement t=(3,)
>>> type(t)         #now this is a tuple
<class 'tuple'>
```

An empty tuple can be created either using a pair of parenthesis or using a function ***tuple()*** as below –

```
>>> t1=()
>>> type(t1)
     <class 'tuple'>

>>> t2=tuple()
>>> type(t2)
     <class 'tuple'>
```

If we provide an argument of type sequence (a list, a string or tuple) to the method ***tuple()***, then a tuple with the elements in a given sequence will be created –

Create tuple using string:

```
>>> t=tuple('Hello')
>>> print(t)
     ('H', 'e', 'l', 'l', 'o')
```

Create tuple using list:

```
>>> t=tuple([3,[12,5],'Hi'])
>>> print(t)
     (3, [12, 5], 'Hi')
```

Create tuple using another tuple:

```
>>> t=('Mango', 34, 'hi')
>>> t1=tuple(t)
>>> print(t1)
     ('Mango', 34, 'hi')
>>> t is t1
     True
```

Note that, in the above example, both t and t1 objects are referring to same memory location. That is, t1 is a reference to t.

Elements in the tuple can be extracted using square-brackets with the help of indices. Similarly, slicing also can be applied to extract required number of items from tuple.

```
>>> t=('Mango', 'Banana', 'Apple')
>>> print(t[1])
     Banana
>>> print(t[1:])
     ('Banana', 'Apple')
>>> print(t[-1])
     Apple
```

Modifying the value in a tuple generates error, because tuples are immutable –

```
>>> t[0]='Kiwi'
TypeError: 'tuple' object does not support item assignment
```

We wanted to replace ‘Mango’ by ‘Kiwi’, which did not work using assignment. But, a tuple can be replaced with another tuple involving required modifications –

```
>>> t=('Kiwi',)+t[1:]
>>> print(t)
     ('Kiwi', 'Banana', 'Apple')
```

5.2.1 Comparing Tuples

Tuples can be compared using operators like >, <, >=, == etc. The comparison happens lexicographically. For example, when we need to check equality among two tuple objects, the first item in first tuple is compared with first item in second tuple. If they are same, 2nd items are compared. The check continues till either a mismatch is found or items get over. Consider few examples –

```
>>> (1,2,3)==(1,2,5)
     False
>>> (3,4)==(3,4)
     True
```

The meaning of < and > in tuples is not exactly *less than* and *greater than***,** instead, it means *comes before* and *comes after.* Hence in such cases, we will get results different from checking equality (==).

```
>>> (1,2,3)<(1,2,5)
     True
>>> (3,4)<(5,2)
     True
```

When we use relational operator on tuples containing non-comparable types, then TypeError will be thrown.

```
>>> (1,'hi')<('hello','world')
TypeError: '<' not supported between instances of 'int' and 'str'
```

The *sort()* function internally works on similar pattern – it sorts primarily by first element, in case of tie, it sorts on second element and so on. This pattern is known as ***DSU*** –

- **Decorate** a sequence by building a list of tuples with one or more sort keys preceding the elements from the sequence,
- **Sort** the list of tuples using the Python built-in *sort*(), and
- **Undecorate** by extracting the sorted elements of the sequence.

Consider a program of sorting words in a sentence from longest to shortest, which illustrates DSU property.

```
txt = 'Ram and Seeta went to forest with Lakshman'
words = txt.split()
t = list()
for word in words:
    t.append((len(word), word))

print('The list is:',t)
t.sort(reverse=True)
res = list()
```

```
    for length, word in t:
        res.append(word)
    print('The sorted list:',res)
```

The output would be –

```
The list is: [(3, 'Ram'), (3, 'and'), (5, 'Seeta'), (4, 'went'),
(2, 'to'), (6, 'forest'), (4, 'with'), (8, 'Lakshman')]

The sorted list: ['Lakshman', 'forest', 'Seeta', 'went', 'with',
'and', 'Ram', 'to']
```

In the above program, we have split the sentence into a list of words. Then, a tuple containing length of the word and the word itself are created and are appended to a list. Observe the output of this list – it is a list of tuples. Then we are sorting this list in descending order. Now for sorting, length of the word is considered, because it is a first element in the tuple. At the end, we extract length and word in the list, and create another list containing only the words and print it.

5.2.2 Tuple Assignment

Tuple has a unique feature of having it at LHS of assignment operator. This allows us to assign values to multiple variables at a time.

```
>>> x,y=10,20
>>> print(x)        #prints 10
>>> print(y)        #prints 20
```

When we have list of items, they can be extracted and stored into multiple variables as below –

```
>>> ls=["hello", "world"]
>>> x,y=ls
>>> print(x)        #prints hello
>>> print(y)        #prints world
```

This code internally means that –

```
x= ls[0]
y= ls[1]
```

The best known example of assignment of tuples is ***swapping two values*** as below –

```
>>> a=10
>>> b=20
>>> a, b = b, a
>>> print(a, b)     #prints 20 10
```

In the above example, the statement `a, b = b, a` is treated by Python as – LHS is a set of variables, and RHS is set of expressions. The expressions in RHS are evaluated and assigned to respective variables at LHS.

Giving more values than variables generates ValueError –

```
>>> a, b=10,20,5
ValueError: too many values to unpack (expected 2)
```

While doing assignment of multiple variables, the RHS can be any type of sequence like list, string or tuple. Following example extracts user name and domain from an email ID.

```
>>> email='chetanahegde@ieee.org'
>>> usrName, domain = email.split('@')
>>> print(usrName)                          #prints chetanahegde
>>> print(domain)                           #prints ieee.org
```

5.2.3 Dictionaries and Tuples

Dictionaries have a method called ***items()*** that returns a list of tuples, where each tuple is a key-value pair as shown below –

```
>>> d = {'a':10, 'b':1, 'c':22}
>>> t = list(d.items())
>>> print(t)
     [('b', 1), ('a', 10), ('c', 22)]
```

As dictionary may not display the contents in an order, we can use ***sort()*** on lists and then print in required order as below –

```
>>> d = {'a':10, 'b':1, 'c':22}
>>> t = list(d.items())
>>> print(t)
     [('b', 1), ('a', 10), ('c', 22)]
>>> t.sort()
>>> print(t)
     [('a', 10), ('b', 1), ('c', 22)]
```

5.2.4 Multiple Assignment with Dictionaries

We can combine the method ***items()*****,** tuple assignment and a for-loop to get a pattern for traversing dictionary:

```
d={'Tom': 1292, 'Jerry': 3501, 'Donald': 8913}
for key, val in list(d.items()):
     print(val,key)
```

The output would be –

```
1292 Tom
3501 Jerry
8913 Donald
```

This loop has two iteration variables because ***items()*** returns a list of tuples. And `key, val` is a tuple assignment that successively iterates through each of the key-value pairs in the dictionary. For each iteration through the loop, both key and value are advanced to the next key-value pair in the dictionary in hash order.

Once we get a key-value pair, we can create a list of tuples and sort them –

```
d={'Tom': 9291, 'Jerry': 3501, 'Donald': 8913}
ls=list()
for key, val in d.items():
    ls.append((val,key))      #observe inner parentheses

print("List of tuples:",ls)
ls.sort(reverse=True)
print("List of sorted tuples:",ls)
```

The output would be –

```
List of tuples: [(9291, 'Tom'), (3501, 'Jerry'), (8913, 'Donald')]
List of sorted tuples: [(9291, 'Tom'), (8913, 'Donald'), (3501,
'Jerry')]
```

In the above program, we are extracting `key, val` pair from the dictionary and appending it to the list `ls`. While appending, we are putting inner parentheses to make sure that each pair is treated as a tuple. Then, we are sorting the list in the descending order. The sorting would happen based on the telephone number (`val`), but not on name (`key`), as first element in tuple is telephone number (`val`).

5.2.5 The Most Common Words

We will apply the knowledge gained about strings, tuple, list and dictionary till here to solve a problem – write a program to find most commonly used words in a text file.

The logic of the program is –

- Open a file
- Take a loop to iterate through every line of a file.
- Remove all punctuation marks and convert alphabets into lower case (Reason explained in Section 3.2.4)
- Take a loop and iterate over every word in a line.
- If the word is not there in dictionary, treat that word as a key, and initialize its value as 1. If that word already there in dictionary, increment the value.
- Once all the lines in a file are iterated, you will have a dictionary containing distinct words and their frequency. Now, take a list and append each key-value (word-frequency) pair into it.
- Sort the list in descending order and display only 10 (or any number of) elements from the list to get most frequent words.

```
import string
fhand = open('test.txt')
counts = dict()
for line in fhand:
    line = line.translate(str.maketrans('', '',string.punctuation))
    line = line.lower()

    for word in line.split():
        if word not in counts:
            counts[word] = 1
        else:
            counts[word] += 1

lst = list()
for key, val in list(counts.items()):
    lst.append((val, key))

lst.sort(reverse=True)
for key, val in lst[:10]:
    print(key, val)
```

Run the above program on any text file of your choice and observe the output.

5.2.6 Using Tuples as Keys in Dictionaries

As tuples and dictionaries are hashable, when we want a dictionary containing composite keys, we will use tuples. For Example, we may need to create a telephone directory where name of a person is Firstname-last name pair and value is the telephone number. Our job is to assign telephone numbers to these keys. Consider the program to do this task –

```
names=(('Tom','Cat'),('Jerry','Mouse'), ('Donald', 'Duck'))
number=[3561, 4014, 9813]

telDir={}

for i in range(len(number)):
    telDir[names[i]]=number[i]

for fn, ln in telDir:
    print(fn, ln, telDir[fn,ln])
```

The output would be –

```
Tom Cat 3561
Jerry Mouse 4014
Donald Duck 9813
```

5.2.7 Summary on Sequences: Strings, Lists and Tuples

Till now, we have discussed different types of sequences viz. strings, lists and tuples. In many situations these sequences can be used interchangeably. Still, due their difference in behavior and ability, we may need to understand pros and cons of each of them and then to decide which one to use in a program. Here are few key points –

1. Strings are more limited compared to other sequences like lists and Tuples. Because, the elements in strings must be characters only. Moreover, strings are immutable. Hence, if we need to modify the characters in a sequence, it is better to go for a list of characters than a string.
2. As lists are mutable, they are most common compared to tuples. But, in some situations as given below, tuples are preferable.
    a. When we have a return statement from a function, it is better to use tuples rather than lists.
    b. When a dictionary key must be a sequence of elements, then we must use immutable type like strings and tuples
    c. When a sequence of elements is being passed to a function as arguments, usage of tuples reduces unexpected behavior due to aliasing.
3. As tuples are immutable, the methods like ***sort()*** and ***reverse()*** cannot be applied on them. But, Python provides built-in functions ***sorted()*** and ***reversed()*** which will take a sequence as an argument and return a new sequence with modified results.