# MODULE III

## 3.1 LISTS

A list is a sequence, Lists are mutable, Traversing a list, List operations, List slices, List Methods, Deleting elements, Lists and functions, Lists and strings, Parsing lines, Objects and values , Aliasing, List arguments, Debugging

## 3.2 DICTIONARIES

Introduction, Dictionary as a set of counters, Dictionaries and files, Looping and Advanced text parsing, Debugging

## 3.3 TUPLES

Tuples are immutable, Comparing tuples, Tuple assignment Dictionaries and tuples, Multiple assignment with dictionaries, The most common words, Using tuples as keys in dictionaries, Sequences: strings, lists, and tuples, Debugging

## 3.4 REGULAR EXPRESSIONS

Character matching in regular expressions, Extracting data using regular expressions, Combining searching and extracting Escape character, Summary, Bonus section for Unix / Linux users

SVIT

# MODULE III

## 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 as *elements* or *items.*
- The elements of 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 same type.
- We can have mixed type of elements in 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 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 (special type of method called as 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]
```

### → Lists are Mutable

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

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

SVIT

- 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 relationship is 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])
        [2,3]
```

➢ 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
```

## → Traversing a List

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

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

SVIT

```
>>> for item in ls:
        print(item)


    34
    hi
    [2,3]
    -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 modification in the elements of list. Hence, referring indices is suitable than referring elements directly.
- The *len()* returns 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 original value by its square).

## → List Operations

- Python allows to use operators + and * on lists.
- The operator + uses two list objects and returns concatenation of those two lists.
- Whereas * operator take 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]
```

## → 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.

SVIT

```
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.

## → List Methods

There are several built-in methods in *list* class for various purposes. Here, we will discuss some 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 list are added at the end of invoking list.

SVIT

```
>>> 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 function will 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 argument as 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 value within 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)
```

SVIT

```
                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 be used 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 comparison cannot 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'
   ```

SVIT

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
```

## → Deleting Elements

Elements can be deleted from a list in different ways. Python provides few built-in methods for 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 give that 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, then this function can be used.

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

SVIT

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 been deleted.

- **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]
```

**Example: Deleting all odd indexed elements of a list –**

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

## → Lists and Functions

- The utility functions like *max(), min(), sum(), len()* etc. can be used on lists.
- Hence most of the 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()*.

## → 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 list of 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 –

SVIT

```
>>> 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.

## → Parsing Lines

- In many situations, we would like to read a file and extract only the lines containing required pattern. 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 stephen.marquard@uct.ac.za Fri Jan 5 09:14:16 2018
From georgek@uct.ac.za Sat Jan 6 06:12:51 2018
………………
```

- 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.

## → Objects and Values

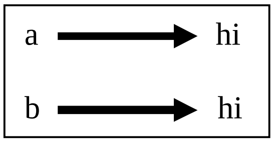
- Whenever we assign two variables with same value, the question arises – whether both the variables are referring to same object, or to different objects.
- This is important aspect to know, because in Python everything is a class object.
- There is nothing like elementary data type.

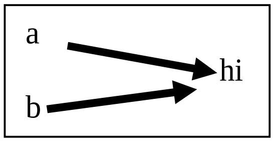
Consider a situation –

```
a= “hi”
b= “hi”
```

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

SVIT





- 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 a same value, they are going to refer 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
```

## → Aliasing

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

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

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

SVIT

in the memory.

- An object with more than one reference has more than one name, hence we say that 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 regards, as they are immutable.

## → List Arguments

- When a list is passed to a function as an argument, then function receives 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']
```

- Here, the argument ls and the parameter t both 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 not going to get anything.
- But, when + operator is applied, t1 remains same but t3 will get the updated result.
- The programmer should understand such differences when he/she creates a function intending to modify a list.
- For example, the following function has no effect on the original list –

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

SVIT

```
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
```

SVIT

```
>>> 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.

SVIT

## → 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 –

SVIT

```
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.

## → 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

SVIT

```
>>> 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).

## → 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.

SVIT

- 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.

## → 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 deletestrparameter 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.

SVIT

## → 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 only the 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 the result 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.

## 3.3TUPLES

- 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!!
```

SVIT

- 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])
```

SVIT

```
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')
```

## → 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.

SVIT

- 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.

### → 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"]
```

SVIT

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

## → 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)]
```

SVIT

## → 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).

## → 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

SVIT

- ➢ Take a loop to iterate through every line of a file.
- ➢ Remove all punctuation marks and convert alphabets into lower case
- ➢ 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.

## → 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)):
```

SVIT

```
          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
```

## → 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.

## → Debugging

- Lists, Dictionaries and Tuples are basically data structures.
- In real-time programming, we may require compound data structures like lists of tuples, dictionaries containing tuples and lists etc.
- But, these compound data structures are prone to ***shape errors*** – that is, errors caused when a data structure has the wrong type, size, composition etc.
- For example, when your code is expecting a list containing single integer, but you are giving a plain integer, then there will be an error.
- When debugging a program to fix the bugs, following are the few things a programmer can try –

  - **Reading:** Examine your code, read it again and check that it says what you meant to say.
  - **Running:** Experiment by making changes and running different versions. Often if you display the right thing at the right place in the program, the problem becomes obvious, but sometimes you have to spend some time to build scaffolding.

SVIT

- **Ruminating:** Take some time to think! What kind of error is it: syntax, runtime, semantic? What information can you get from the error messages, or from the output of the program? What kind of error could cause the problem you're seeing? What did you change last, before the problem appeared?
- **Retreating:** At some point, the best thing to do is back off, undoing recent changes, until you get back you can start rebuilding.

## 3.4 REGULAR EXPRESSIONS

- Searching for required patterns and extracting only the lines/words matching the pattern is a very common task in solving problems programmatically.
- We have done such tasks earlier using string slicing and string methods like *split()*, *find()* etc.
- As the task of searching and extracting is very common, Python provides a powerful library called ***regular expressions*** to handle these tasks elegantly.
- Though they have quite complicated syntax, they provide efficient way of searching the patterns.
- The regular expressions are themselves little programs to search and parse strings.
- To use them in our program, the library/module ***re*** must be imported.
- There is a ***search()*** function in this module, which is used to find particular substring within a string.
- Consider the following example –

```
import re
fhand = open('myfile.txt')
for line in fhand:
        line = line.rstrip()
        if re.search('how', line):
                print(line)
```

- By referring to file *myfile.txt* that has been discussed in previous Chapters, the output would be

```
 hello, how are you?
how about you?
```

- In the above program, the *search()* function is used to search the lines containing a word *how*.
- One can observe that the above program is not much different from a program that uses ***find()*** function of strings. But, regular expressions make use of special characters with specific meaning.
- In the following example, we make use of caret (^) symbol, which indicates beginning of the line.

```
import re
hand = open('myfile.txt')
for line in hand:
        line = line.rstrip()
        if re.search('^how', line):
                print(line)
```

**The output would be –**

```
how about you?
```

- Here, we have searched for a line which starts with a string *how.*
- Again, this program will not makes use of regular expression fully.
- Because, the above program would have been written using a string function *startswith()*. Hence,

SVIT

in the next section, we will understand the true usage of regular expressions.

## → Character Matching in Regular Expressions

- Python provides a list of meta-characters to match search strings.
- Table below shows the details of few important metacharacters.
- Some of the examples for quick and easy understanding of regular expressions are given in next Table.

**Table : List of Important Meta-Characters**

| Character | Meaning |
|---|---|
| ^ (caret) | Matches beginning of the line |
| $ | Matches end of the line |
| . (dot) | Matches any single character except newline. Using option *m*, then newline also can be matched |
| […] | Matches any single character in brackets |
| [^…] | Matches any single character NOT in brackets |
| re* | Matches 0 or more occurrences of preceding expression. |
| re+ | Matches 1 or more occurrence of preceding expression. |
| re? | Matches 0 or 1 occurrence of preceding expression. |
| re{ n} | Matches exactly n number of occurrences of preceding expression. |
| re{ n,} | Matches n or more occurrences of preceding expression. |
| re{ n, m} | Matches at least n and at most m occurrences of preceding expression. |
| a\| b | Matches either a or b. |
| (re) | Groups regular expressions and remembers matched text. |
| \d | Matches digits. Equivalent to [0-9]. |
| \D | Matches non-digits. |
| \w | Matches word characters. |
| \W | Matches non-word characters. |
| \s | Matches whitespace. Equivalent to [\t\n\r\f]. |
| \S | Matches non-whitespace. |
| \A | Matches beginning of string. |
| \Z | Matches end of string. If a newline exists, it matches just before newline. |
| \z | Matches end of string. |
| \b | Matches the empty string, but only at the start or end of a word. |
| \B | Matches the empty string, but not at the start or end of a word. |
| ( ) | When parentheses are added to a regular expression, they are ignored for the purpose of matching, but allow you to extract a particular subset of the matched string rather than the whole string when using findall() |

**Table : Examples for Regular Expressions**

| Expression | Description |
|---|---|
| [Pp]ython | Match "Python" or "python" |

SVIT

| | |
|---|---|
| rub[ye] | Match "ruby" or "rube" |
| [aeiou] | Match any one lowercase vowel |
| [0-9] | Match any digit; same as [0123456789] |
| [a-z] | Match any lowercase ASCII letter |
| [A-Z] | Match any uppercase ASCII letter |
| [a-zA-Z0-9] | Match any of uppercase, lowercase alphabets and digits |
| [^aeiou] | Match anything other than a lowercase vowel |
| [^0-9] | Match anything other than a digit |

- Most commonly used metacharacter is dot, which matches any character.
- Consider the following example, where the regular expression is for searching lines which starts with I and has any two characters (any character represented by two dots) and then has a character m.

```
import re
fhand = open('myfile.txt')
for line in fhand:
    line = line.rstrip()
    if re.search('^I..m', line):
        print(line)
```

**The output would be –**

```
I am doing fine.
```

- Note that, the regular expression ^I..m not only matches 'I am', but it can match 'Isdm', 'I*3m' and so on.
- That is, between Iand m, there can be any two characters.
- In the previous program, we knew that there are exactly two characters between I and m. Hence, we could able to give two dots.
- But, when we don't know the exact number of characters between two characters (or strings), we can make use of dot and + symbols together.
- Consider the below given program –

```
import re
hand = open('myfile.txt')
for line in hand:
    line = line.rstrip()
    if re.search('^h.+u', line):
        print(line)
```

**The output would be –**

```
hello, how are you?
how about you?
```

- Observe the regular expression ^h.+u here.
- It indicates that, the string should be starting with h and ending with u and there may by any number of (dot and +) characters in- between.

SVIT

**Few examples:**

- To understand the behavior of few basic meta characters, we will see some examples.
- The file used for these examples is *mbox-short.txt* which can be downloaded from – https://www.py4e.com/code3/mbox-short.txt
- Use this as input and try following examples –

- **Pattern to extract lines starting with the word *From* (or *from*) and ending with *edu*:**

```
import re
fhand = open('mbox-short.txt')
 for line in fhand:
        line =  line.rstrip()
        pattern = ‘^[Ff]rom.*edu$’
        if re.search(pattern,  line):
                print(line)
```

Here the pattern given for regular expression indicates that the line should start with either *From* or *from*. Then there may be 0 or more characters, and later the line should end with *edu*.

- **Pattern to extract lines ending with any digit:**

Replace the pattern by following string, rest of the program will remain the same.

```
pattern = ‘[0-9]$’
```

- **Using *Not* :**

```
pattern = ‘^[^a-z0-9]+’
```

Here, the first ^ indicates we want something to match in the beginning of a line. Then, the ^ inside square-brackets indicate *do not match any single character within bracket*. Hence, the whole meaning would be – line must be started with anything other than a lower-case alphabets and digits. In other words, the line should not be started with lowercase alphabet and digits.

- **Start with upper case letters and end with digits:**

```
pattern = '^[A-Z].*[0-9]$'
```

Here, the line should start with capital letters, followed by 0 or more characters, but must end with any digit.

## → Extracting Data using Regular Expressions

- Python provides a method ***findall()*** to extract all of the substrings matching a regular expression.
- This function returns a list of all non-overlapping matches in the string.
- If there is no match found, the function returns an empty list.
- Consider an example of extracting anything that looks like an email address from any line.

```
import re
s = 'A message from csev@umich.edu to cwen@iupui.edu about meeting @2PM'
```

SVIT

```
lst = re.findall('\S+@\S+', s)
print(lst)
```

The output would be –

```
['csev@umich.edu', 'cwen@iupui.edu']
```

- Here, the pattern indicates at least one non-white space characters (\S) before @ and at least one non-white space after @.
- Hence, it will not match with @2pm, because of a white- space before @.
- Now, we can write a complete program to extract all email-ids from the file.

```
import re
fhand = open('mbox-short.txt')
for line in fhand:
        line = line.rstrip()
        x = re.findall('\S+@\S+', line)
        if len(x) > 0:
                print(x)
```

- Here, the condition len(x) > 0 is checked because, we want to print only the line which contain an email-ID. If any line do not find the match for a pattern given, the *findall()* function will return an empty list. The length of empty list will be zero, and hence we would like to print the lines only with length greater than 0.

**The output of above program will be something as below –**

```
['stephen.marquard@uct.ac.za'] ['<postmaster@collab.sakaiproject.org>']
['<200801051412.m05ECIaH010327@nakamura.uits.iupui.edu>']
['<source@collab.sakaiproject.org>;'] ['<source@collab.sakaiproject.org>;']
['<source@collab.sakaiproject.org>;'] ['apache@localhost)']
……………………………….
………………………………..
```

- Note that, apart from just email-ID's, the output contains additional characters (<, >, ; etc) attached to the extracted pattern. To remove all that, refine the pattern. That is, we want email-ID to be started with any alphabets or digits, and ending with only alphabets. Hence, the statement would be –

```
x = re.findall('[a-zA-Z0-9]\S*@\S*[a-zA-Z]', line)
```

## → Combining Searching and Extracting

- Assume that we need to extract the data in a particular syntax.
- For example, we need to extract the lines containing following format –

```
X-DSPAM-Confidence: 0.8475
X-DSPAM-Probability: 0.0000
```

SVIT

- The line should start with X-, followed by 0 or more characters. Then, we need a colon and white-space. They are written as it is.
- Then there must be a number containing one or more digits with or without a decimal point. Note that, we want dot as a part of our pattern string, but not as meta character here. The pattern for regular expression would be –

```
^X-.*: [0-9.]+
```

The complete program is –

```
import re
hand = open('mbox-short.txt')
for line in hand:
    line = line.rstrip()
    if re.search('^X\S*: [0-9.]+', line):
        print(line)
```

**The output lines will as below –**

```
X-DSPAM-Confidence: 0.8475
X-DSPAM-Probability: 0.0000
X-DSPAM-Confidence: 0.6178
X-DSPAM-Probability: 0.0000
X-DSPAM-Confidence: 0.6961
X-DSPAM-Probability: 0.0000
…………………………………………………
…………………………………………………
```

- Assume that, we want only the numbers (representing confidence, probability etc) in the above output.
- We can use *split()* function on extracted string. But, it is better to refine regular expression. To do so, we need the help of parentheses.
- When we add parentheses to a regular expression, they are ignored when matching the string. But when we are using ***findall()***, parentheses indicate that while we want the whole expression to match, we only are interested in extracting a portion of the substring that matches the regular expression.

```
import re
hand = open('mbox-short.txt')
for line in hand:
    line = line.rstrip()
    x = re.findall('^X-\S*: ([0-9.]+)', line)
    if len(x) > 0:
        print(x)
```

- Because of the parentheses enclosing the pattern above, it will match the pattern starting with X- and extracts only digit portion. Now, the output would be –

```
['0.8475']
['0.0000']
['0.6178']
['0.0000']
```

SVIT

```
['0.6961']
…………………
………………..
```

- Another example of similar form: The file *mbox-short.txt* contains lines like –

  Details: http://source.sakaiproject.org/viewsvn/?view=rev&rev=39772
- We may be interested in extracting only the revision numbers mentioned at the end of these lines. Then, we can write the statement –

```
x = re.findall('^Details:.*rev=([0-9.]+)', line)
```

- The regex here indicates that the line must start with Details:, and has something with rev= and then digits.
- As we want only those digits, we will put parenthesis for that portion of expression.
- Note that, the expression [0-9] is greedy, because, it can display very large number. It keeps grabbing digits until it finds any other character than the digit.
- The output of above regular expression is a set of revision numbers as given below –

```
['39772']
['39771']
['39770']
['39769']
………………………
………………………
```

- Consider another example – we may be interested in knowing time of a day of each email. The file *mbox-short.txt* has lines like –

  From stephen.marquard@uct.ac.za Sat Jan 5 09:14:16 2008

- Here, we would like to extract only the hour 09. That is, we would like only two digits representing hour. Hence, we need to modify our expression as –

```
x = re.findall('^From .* ([0-9][0-9]):', line)
```

- Here, [0-9][0-9] indicates that a digit should appear only two times.
- The alternative way of writing this would be -

```
x = re.findall('^From .* ([0-9]{2}):', line)
```

- The number 2 within flower-brackets indicates that the preceding match should appear exactly two times.
- Hence [0-9]{2} indicates there can be exactly two digits.
- Now, the output would be –

```
['09']
['18']
['16']
['15']
…………………
…………………
```

## → Escape Character

- As we have discussed till now, the character like dot, plus, question mark, asterisk, dollar etc. are meta characters in regular expressions.

SVIT

- Sometimes, we need these characters themselves as a part of matching string.
- Then, we need to escape them using a back- slash.
- For example,

```
import re
x = 'We just received $10.00 for cookies.'
y = re.findall('\$[0-9.]+',x)
```

**Output:**

```
['$10.00']
```

- Here, we want to extract only the price $10.00. As, $ symbol is a metacharacter, we need to use \ before it.
- So that, now $ is treated as a part of matching string, but not as metacharacter.

## → Bonus Section for Unix/Linux Users

- Support for searching files using regular expressions was built into the Unix OS.
- There is a command-line program built into Unix called ***grep*** (Generalized Regular Expression Parser) that behaves similar to ***search()*** function.

```
$ grep '^From:'          mbox-short.txt
```

**Output:**

```
From: stephen.marquard@uct.ac.za From:
louis@media.berkeley.edu From:
zqian@umich.edu
From: rjlowe@iupui.edu
```

- Note that, *grep* command does not support the non-blank character \S, hence we need to use [^ ]indicating not a white-space.

SVIT