5.1 Lists

Data Structure

- A data structure is a collection of elements that are stored in some way. The most commonly used data structure in Python is the sequence.
- Lists are the sequential data structures, as each element in it is placed in a sequential manner.
- List is a collection of items (strings or integers or float values or other lists)
- Lists are enclosed in [].
- Each item in the list has an assigned index value and we can directly access the elements using their respective indexes.
- Each item in the list is separated by commas and the order in which the items are stored is the same as the order in which they are inserted.
- Lists are mutable. It means the items in a list can be modified as per our requirements.

List Creation

Below example was discussed at the timestamp 1:22 in the video.

```
emptyList = []

lst = ['one', 'two', 'three', 'four'] # list of strings

lst2 = [1, 2, 3, 4] #list of integers

lst3 = [[1, 2], [3, 4]] # list of lists

lst4 = [1, 'ramu', 24, 1.24] # list of different datatypes

print(lst4)
```

```
[1, 'ramu', 24, 1.24]
```

We can create an empty list either using a pair of empty square braces (ie., []) or using the list() function without any arguments. (ie., list())
We can have elements belonging to different data types in a list.

List Length

We use the **len()** function to find out the length of a list.

The below example was discussed at the timestamp 2:45

```
lst = ['one', 'two', 'three', 'four']

#find length of a list
print(len(lst))
```

```
4
```

List Insert

Below example was discussed at the timestamp 3:38 in the video.

```
#syntax: lst.insert(x, y)

lst = ['one', 'two', 'four']

lst.insert(2, "three") # will add element y at location x

print(lst)
```

```
['one', 'two', 'three', 'four']
```

We use the **insert()** function with 2 parameters to insert elements into the list. The first argument would be the index position at which the element/value has to be inserted and the second argument would be the value which has to be inserted.

List Remove

Below example was discussed at the timestamp 4:56 in the video.

```
#syntax: lst.remove(x)

lst = ['one', 'two', 'three', 'four', 'two']

lst.remove('two') #it will remove first occurence of 'two' in a given list

print(lst)
```

```
['one', 'three', 'four', 'two']
```

- We use the **remove()** method to remove an element from the given list.
- In case, if the specified element is present multiple times in the given list, then it only removes the first occurrence of the given element.

➢ In case, if the specified element is not at all present in the given list, then it throws an error.

List append and Extend

Below examples are discussed at the timestamp 3:13 and 6:00 respectively in the video.

```
lst = ['one', 'two', 'three', 'four']

lst.append('five') # append will add the item at the end

print(lst)
```

```
['one', 'two', 'three', 'four', 'five']
```

```
lst = ['one', 'two', 'three', 'four']

lst2 = ['five', 'six']

#append
lst.append(lst2)

print(lst)
```

```
['one', 'two', 'three', 'four', ['five', 'six']]
```

```
lst = ['one', 'two', 'three', 'four']

lst2 = ['five', 'six']

#extend will join the list with list1

lst.extend(lst2)

print(lst)
```

```
['one', 'two', 'three', 'four', 'five', 'six']
```

➢ Whenever we want to append only one element, then we can use either **append()** or **extend()**. Both work the same way.

➢ But if we want to append one list with another list, then if we use **append()**, then the second list as it is in the form of a list gets appended to the first list. Whereas if we use **extend()**, all the elements in the second list would be appended one after the other to the first list.

List Delete

Below examples are discussed at the timestamp 7:47 in the video.

```
#del to remove item based on index position

lst = ['one', 'two', 'three', 'four', 'five']

del lst[1]
print(lst)

#or we can use pop() method
a = lst.pop(1)
print(a)

print(lst)
```

```
['one', 'three', 'four', 'five']
three
['one', 'four', 'five']
```

```
lst = ['one', 'two', 'three', 'four']

#remove an item from list
lst.remove('three')

print(lst)
```

```
['one', 'two', 'four']
```

We can remove any element using the **pop()** method. Whenever we use the **pop()** method, then we have to pass the index whose value has to be removed.

We can also remove the elements using the **del** keyword. An example would be **del a[2]** removes the index present at 2nd index position in the list 'a'.

List related keywords in Python

Below is an example discussed at the timestamp 10:00.

```
#keyword 'in' is used to test if an item is in a list
lst = ['one', 'two', 'three', 'four']

if 'two' in lst:
    print('AI')

#keyword 'not' can combined with 'in'
if 'six' not in lst:
    print('ML')
```

```
AI
ML
```

The **'in'** and **'not in'** operators are the membership operators used to check whether the the given elements are members of the given list.

List Reverse

The below example begins at the timestamp 11:15 in the video.

```
|: #reverse is reverses the entire list

   lst = ['one', 'two', 'three', 'four']

   lst.reverse()

   print(lst)
```

```
['four', 'three', 'two', 'one']
```

We use the **reverse()** method to reverse the given list. This operation is performed on the input itself. It doesn't return a copy.

List Sorting

- The easiest way to sort a given list is using the **sorted()** function. The other way is to sort using the **sort()** method.
- The main difference between both of them is that **sorted()** function returns a copy of the sorted form of a list (It doesn't sort the given list) whereas the **sort()** method performs the sort operation on the given list itself. It doesn't return any copy.
- When we use the **sort()** function the original list is changed whereas when we use the **sorted()** function, the original list doesn't change.
- Whenever we want to sort a list, we need to make sure all the elements in the list belong to the same data type.

➢ If we want to sort the list in descending order either using sorted() function or sort() method, then we have to use the **reverse = True** argument.

The below examples are discussed starting from the timestamp 11:25.

```
#create a list with numbers
numbers = [3, 1, 6, 2, 8]

sorted_lst = sorted(numbers)


print("Sorted list :", sorted_lst)

#original list remain unchanged
print("Original list: ", numbers)
```

```
Sorted list : [1, 2, 3, 6, 8]
Original list:  [3, 1, 6, 2, 8]
```

```
#print a list in reverse sorted order
print("Reverse sorted list :", sorted(numbers, reverse=True))

#orginal list remain unchanged
print("Original list :",  numbers)
```

```
Reverse sorted list : [8, 6, 3, 2, 1]
Original list : [3, 1, 6, 2, 8]
```

```
lst = [1, 20, 5, 5, 4.2]

#sort the list and stored in itself
lst.sort()

# add element 'a' to the list to show an error

print("Sorted list: ", lst)
```

```
Sorted list:  [1, 4.2, 5, 5, 20]
```

```
lst = [1, 20, 'b', 5, 'a']
print(lst.sort()) # sort list with element of different datatypes.
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-18-98d08ff0e3ba> in <module>()
      1 lst = [1, 20, 'b', 5, 'a']
----> 2 print(lst.sort())

TypeError: '<' not supported between instances of 'str' and 'int'
```

List having multiple references

The below example was discussed at the timestamp 15:10

```
lst = [1, 2, 3, 4, 5]
abc = lst
abc.append(6)

#print original list
print("Original list: ", lst)
```

```
Original list:  [1, 2, 3, 4, 5, 6]
```

We can use multiple references for the same object. All these multiple references, point to the same object. If we make any changes using any of these references, the original object gets affected.

String Split to create a list

The below example was discussed at the timestamp 16:40 in the video.

```
#let's take a string

s = "one,two,three,four,five"
slst = s.split(',')
print(slst)
```

```
['one', 'two', 'three', 'four', 'five']
```

```
s = "This is applied AI Course"
split_lst = s.split() # default split is white-character: space or tab
print(split_lst)
```

```
['This', 'is', 'applied', 'AI', 'Course']
```

We use the split() method to split the given string into a list of strings. If we want to split the given input string on the basis of any alphabet or number or a string pattern or any special character, then we have to enclose it within a single quote and then pass it as an argument in the split() method.

- For example, let us assume the input string is **s1 = 'Welcome to the 1st session of Machine Learning and Deep Learning'**
- In this example, if we want to split this string with the number '1' as the separator, then the syntax for it would be s1.split('1').
- If we want to split this string with the substring 'of' as the separator, then the syntax for it would be s1.split('of').

- ➢ You also can use special characters like comma(,), colon(:), semi-colon(‘;), etc as the separators, but these separators should be present in the given input string.
- ➢ If we do not specify the separator, then the whiltespace would be considered as the default separator.

List Indexing

- ➢ We can access the elements of a string using the indexes. The indexes start from **0** and the last value would be **(length of the string-1)**.
- ➢ Python also supports negative indexing which means traversing in the reverse direction.

Below is an example discussed at the timestamp 18:15

```
lst = [1, 2, 3, 4]
print(lst[1]) #print second element

#print last element using negative index
print(lst[-2])
```

```
2
3
```

List Slicing

- ➢ Accessing parts of segments is called slicing.
- ➢ The key point to remember is that the end value represents the first value that is not in the selected slice.

Below example is discussed at the timestamp 19:50

```
numbers = [10, 20, 30, 40, 50,60,70,80]

#print all numbers
print(numbers[:])

#print from index 0 to index 3
print(numbers[0:4])
```

```
[10, 20, 30, 40, 50, 60, 70, 80]
[10, 20, 30, 40]
```

```
print (numbers)
#print alternate elements in a list
print(numbers[::2])


#print elemnts start from 0 through rest of the list
print(numbers[2::2])
```

```
[10, 20, 30, 40, 50, 60, 70, 80]
[10, 30, 50, 70]
[30, 50, 70]
```

List extend using the '+' operator

The other way to perform the **extend()** operator is using the '+' operator. Using the '+' operator, we can concatenate two lists.
The below example was discussed at the timestamp 23:25.

```
lst1 = [1, 2, 3, 4]
lst2 = ['varma', 'naveen', 'murali', 'brahma']
new_lst = lst1 + lst2

print(new_lst)
```

```
[1, 2, 3, 4, 'varma', 'naveen', 'murali', 'brahma']
```

List Count

We use the **count()** method to find out the number of occurrences of a given element in the given list.
The below example was discussed at the timestamp 24:10

```
numbers = [1, 2, 3, 1, 3, 4, 2, 5]

#frequency of 1 in a list
print(numbers.count(1))

#frequency of 3 in a list
print(numbers.count(3))
```

```
2
2
```

List Looping

We can loop through the entire list by considering each element one by one using a ‘for’ loop. The below example was discussed at the timestamp 25:25

```
#loop through a list

lst = ['one', 'two', 'three', 'four']

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

```
one
two
three
four
```

List Comprehensions

- ➢ List comprehensions provide a concise way to create lists.
- ➢ Common applications are to make new lists where each element is the result of some operations applied to each member of another sequence or iterable, or to create a subsequence of those elements that satisfy a certain condition.

The below example was discussed at the timestamp 26:40

```
# without list comprehension
squares = []
for i in range(10):
    squares.append(i**2)   #list append
print(squares)
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

```
#using list comprehension
squares = [i**2 for i in range(10)]
print(squares)
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

```
#example

lst = [-10, -20, 10, 20, 50]

#create a new list with values doubled
new_lst = [i*2 for i in lst]
print(new_lst)

#filter the list to exclude negative numbers
new_lst = [i for i in lst if i >= 0]
print(new_lst)


#create a list of tuples like (number, square_of_number)
new_lst = [(i, i**2) for i in range(10)]
print(new_lst)
```

```
[-20, -40, 20, 40, 100]
[10, 20, 50]
[(0, 0), (1, 1), (2, 4), (3, 9), (4, 16), (5, 25), (6, 36), (7, 49), (8, 64), (9, 81)]
```

Nested List Comprehensions

The below example was discussed at the timestamp 31:40

```python
#let's suppose we have a matrix

matrix = [
    [1, 2, 3, 4],
    [5, 6, 7, 8],
    [9, 10, 11, 12]
]

#transpose of a matrix without list comprehension
transposed = []
for i in range(4):
    lst = []
    for row in matrix:
        lst.append(row[i])
    transposed.append(lst)

print(transposed)
```

```
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

```python
#with list comprehension
transposed = [[row[i] for row in matrix] for i in range(4)]
print(transposed)
```

```
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

5.2 Tuples (Part 1)

- Tuple is a data structure that is similar to a List.
- The only difference is that Lists are mutable whereas Tuples are immutable. It means, in a tuple, once after we assign the elements, they cannot be changed whereas in a List we can change whenever needed.

Tuple Creation

The below examples were discussed starting from the timestamp 1:10 in the video.

```
: #empty tuple
t = ()

#tuple having integers
t = (1, 2, 3)
print(t)

#tuple with mixed datatypes
t = (1, 'raju', 28, 'abc')
print(t)

#nested tuple
t = (1, (2, 3, 4), [1, 'raju', 28, 'abc'])
print(t)
```

```
(1, 2, 3)
(1, 'raju', 28, 'abc')
(1, (2, 3, 4), [1, 'raju', 28, 'abc'])
```

Similar to a list, even a tuple can contain elements belonging to different data types. Also the tuples support indexing.

```
#only parenthesis is not enough
t = ('satish')
type(t)
```

```
str
```

```
#need a comma at the end
t = ('satish',)
type(t)
```

```
tuple
```

We cannot manipulate/modify the elements present in a tuple. But if there are any mutable objects(like list) present as an element in a tuple, we could not completely remove that mutable object, but we can manipulate the elements in that mutable object.

In general, we define a tuple using (). But if there is only 1 element in a tuple, then it has to be followed by a comma(,) in the tuple. Otherwise, the interpreter considers the data type, as the data type of that object, but not as a tuple object.

In the above example, we see
t = ('satish')
This example, even though we have enclosed the element in (), as there is only one element and it is a string, the interpreter considers 't' as a string object, but not as a tuple. If we want to make the interpreter consider this object as a tuple, then the syntax should be
t = ('satish',)
Note: Whenever the tuple we define consists of only 1 element, then it always has to be followed by a comma.

Accessing Elements in a Tuple

Tuple also supports indexing and slicing similar to a list.

```
]: t = ('satish', 'murali', 'naveen', 'srinu', 'brahma')

   print(t[1])
```

```
murali
```

```
]: #negative index
   print(t[-1]) #print last element in a tuple
```

```
brahma
```

```
]: #nested tuple
   t = ('ABC', ('satish', 'naveen', 'srinu'))

   print(t[1])
```

```
('satish', 'naveen', 'srinu')
```

```
]: print(t[1][2])
```

```
srinu
```

```
]: #Slicing
   t = (1, 2, 3, 4, 5, 6)

   print(t[1:4])

   #print elements from starting to 2nd last elements
   print(t[:-2])

   #print elements from starting to end
   print(t[:])
```

```
(2, 3, 4)
(1, 2, 3, 4)
(1, 2, 3, 4, 5, 6)
```

Changing the elements in a tuple

The below examples are discussed from the timestamp 7:22

```
: #creating tuple
  t = (1, 2, 3, 4, [5, 6, 7])

  t[2] = 'x' #will get TypeError
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-3-9f4cbf6ee0de> in <module>()
      2 t = (1, 2, 3, 4, [5, 6, 7])
      3
----> 4 t[2] = 'x' #will get TypeError

TypeError: 'tuple' object does not support item assignment
```

As tuples are immutable objects, one cannot modify the elements present in them. But we can modify the values if an object present in a tuple is a mutable object (like a list). Below is an example of it.

```
#creating tuple
t = (1, 2, 3, 4, [5, 6, 7])

t[2] = 'x' #will get TypeError
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-3-9f4cbf6ee0de> in <module>()
      2 t = (1, 2, 3, 4, [5, 6, 7])
      3
----> 4 t[2] = 'x' #will get TypeError

TypeError: 'tuple' object does not support item assignment
```

```
t[4][1] = 'satish'
print(t)
```

```
(1, 2, 3, 4, [5, 'satish', 7])
```

Concatenation of Tuples

We can perform concatenation of tuples, but the concatenation returns a copy. We cannot perform concatenation operations in place for a tuple.

```
#concatinating tuples

t = (1, 2, 3) + (4, 5, 6)
print(t)
```

```
(1, 2, 3, 4, 5, 6)
```

In the above example, we are concatenating the tuple (4,5,6) to the tuple (1,2,3) and are storing the result in 't'.
If we want to create a tuple with a repetition of a single value, then we have to use the '*' operator.

```
#repeat the elements in a tuple for a given number of times using the * operator.
t = (('satish',) * 4)
print(t)
```

```
('satish', 'satish', 'satish', 'satish')
```

In the above example, we want to create a tuple with the string 'satish' occurring 4 times. So we are first creating a tuple with the string 'satish' occurring only once and then multiplying it with the number 4 using the '*' operator.

5.3 Tuples (Part 2)

Deletion of a Tuple

We have to use the **del** keyword to delete any python object. The same keyword we can use to delete a tuple.
The below example is discussed from the timestamp 0:05

```
: #we cannot change the elements in a tuple.
  # That also means we cannot delete or remove items from a tuple.

  #delete entire tuple using del keyword
  t = (1, 2, 3, 4, 5, 6)

  #delete entire tuple
  del t
```

Tuple Count

The number of occurrences of the given element in a given tuple is obtained using the **count()** function.
The below example was discussed starting from the timestamp 0:40

```
: t = (1, 2, 3, 1, 3, 3, 4, 1)

  #get the frequency of particular element appears in a tuple
  t.count(1)
```

```
: 3
```

In this example, we are checking how many times the number 1 is occurring in the given tuple. As it is present 3 times, the result is obtained as 3.

Finding the index in a tuple

We use the **index()** method to find out the index of a particular element in a given tuple. If the given element occurs multiple times in a given tuple, then it returns the index of the first occurrence of that element in the given tuple.
The below example was discussed starting from the timestamp 1:05

```
t = (1, 2, 3, 1, 3, 3, 4, 1)

print(t.index(3)) #return index of the first element is equal to 3

#print index of the 1
```

```
2
```

In this example, as the number 3 is occurring 3 times, the index() method returns the index of the first occurrence of 3.
If the given element is not present in the tuple, then it throws an error.

Tuple Membership

We can check if a given element is present in a tuple, using the '**in**' keyword.
The below example is discussed starting from the timestamp 1:45

```
#test if an item exists in a tuple or not, using the keyword in.
t = (1, 2, 3, 4, 5, 6)

print(1 in t)
```

```
True
```

```
print(7 in t)
```

```
False
```

In the first example, we are checking if the number 1 is present in the tuple 't'. As the value is present, it is returning True.
In the second example, we are checking if the number 7 is present in the tuple 't'. As the value is not present, it is returning False.

Built-in Functions

Tuple Length

We use the **len()** function to find out the number of elements present in the given tuple.

```
t = (1, 2, 3, 4, 5, 6)
print(len(t))
```

```
6
```

Tuple Sort

We use the **sorted()** function to return the sorted form of a tuple. The **sorted()** function doesn't perform the sort operation in-place. It returns a copy of the sorted form only in the form of a list.

```
t = (4, 5, 1, 2, 3)

new_t = sorted(t)
print(new_t) #Take elements in the tuple and return a new sorted list
             #(does not sort the tuple itself).
```

```
[1, 2, 3, 4, 5]
```

Maximum element

We use **max()** function to return the maximum value from the given tuple.

```
#get the largest element in a tuple
t = (2, 5, 1, 6, 9)

print(max(t))
```

```
9
```

Minimum element

We use the **min()** function to return the minimum value from the given tuple.

```
#get the smallest element in a tuple
print(min(t))
```

```
1
```

.

5.4 Sets

- ➢ A set is an unordered collection of items. Every element is unique (no duplicates).
- ➢ A set is a mutable data structure. We can add or remove items from it.
- ➢ Sets can be used to perform mathematical set operations like union, intersection, symmetric difference, etc
- ➢ The order in which the elements are inserted into a set is sometimes different from the order in which they are displayed.

Set Creation

Below are the examples of different ways in which we can create a set. These examples are discussed in the video starting from the timestamp 0:55.

```
#set of integers
s = {1, 2, 3}
print(s)

#print type of s
print(type(s))
```

```
set([1, 2, 3])
<type 'set'>
```

```
#set doesn't allow duplicates. They store only one instance.
s = {1, 2, 3, 1, 4}
print(s)
```

```
{1, 2, 3, 4}
```

```
#we can make set from a list
s = set([1, 2, 3, 1])
print(s)
```

```
{1, 2, 3}
```

```
#initialize a set  with set() method
s = set()

print(type(s))
```

```
<class 'set'>
```

If we want to create an empty set, then the syntax for it would be

s = set()

If we just use the below example, then it creates a dictionary object, but not a set object.

s = {}

As a set doesn't allow duplicate elements, even if we pass duplicate values while creating a set, in the final result, we'll get them only once.

We can use elements of different data types in a set and we can modify the values in a set, as it is a mutable object.

Addition of elements to a set

Below examples are discussed in the video starting from the timestamp 2:45.

```
#we can add single element using add() method and
#add multiple elements using update() method
s = {1, 3}

#set object doesn't support indexing
print(s[1]) #will get TypeError
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-4-c52fc339e293> in <module>()
      4
      5 #set object doesn't support indexing
----> 6 print(s[1]) #will get TypeError

TypeError: 'set' object does not support indexing
```

```
#add element
s.add(2)
print(s)
```

```
{1, 2, 3}
```

```
#add multiple elements
s.update([5, 6, 1])
print(s)
```

```
{1, 2, 3, 5, 6}
```

```
#add list and set
s.update([8, 9], {10, 2, 3})
print(s)
```

```
{1, 2, 3, 5, 6, 8, 9, 10}
```

If we want to add the elements one at a time, then we have to use the **add()** method, whereas if we want to add more than one element at a time, then we have to use the **update()** method.

We can use integers, float values, strings, lists, tuples or combination of these objects as the elements in a set.

Removal of elements from a set

Below are the examples that were discussed in the video starting from the timestamp 5:20.

```
#A particular item can be removed from set using methods,
#discard() and remove().

s = {1, 2, 3, 5, 4}
print(s)

s.discard(4)    #4 is removed from set s

print(s)
```

```
{1, 2, 3, 4, 5}
{1, 2, 3, 5}
```

```
#remove an element
s.remove(2)

print(s)
```

```
{1, 3, 5}
```

```
#remove an element not present in a set s
s.remove(7) # will get KeyError
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-14-f37cc9806699> in <module>()
      1 #remove an element not present in a set s
----> 2 s.remove(7) # will get KeyError

KeyError: 7
```

```
#discard an element not present in a set s
s.discard(7)
print(s)
```

```
{1, 3, 5}
```

```
#we can remove item using pop() method

s = {1, 2, 3, 5, 4}

s.pop() #remove random element

print(s)
```

```
{2, 3, 4, 5}
```

```
s.pop()
print(s)
```

```
{3, 4, 5}
```

```
s = {1, 5, 2, 3, 6}

s.clear()   #remove all items in set using clear() method

print(s)
```

```
set()
```

If we want to remove any random element from the set, then we have to use the **pop()** method. Generally **pop()** throws an error only when the given set is empty.
If we want to remove any particular element from the set, then we have to use either **remove()** or **discard()** methods.
The main difference between **remove()** and **discard()** is that if any element that has to be removed from the set is not present, then the **remove()** method throws an error whereas the **discard()** method doesn't throw any error.
If we want to remove all the elements at a time from the set, then we have to use the **clear()** method.

Python Set Operations

We majorly perform operations like union, intersection, difference and symmetric difference between the sets.
Below are the examples discussed starting from the timestamp 7:50.

```
set1 = {1, 2, 3, 4, 5}
set2 = {3, 4, 5, 6, 7}

#union of 2 sets using | operator

print(set1 | set2)
```

```
{1, 2, 3, 4, 5, 6, 7}
```

```
#another way of getting union of 2 sets
print(set1.union(set2))
```

```
{1, 2, 3, 4, 5, 6, 7}
```

```
#intersection of 2 sets using & operator
print(set1 & set2)
```

```
{3, 4, 5}
```

```
#use intersection function
print(set1.intersection(set2))
```

```
{3, 4, 5}
```

```
#set Difference: set of elements that are only in set1 but not in set2

print(set1 - set2)
```

```
{1, 2}
```

```
#use differnce function
print(set1.difference(set2))
```

```
{1, 2}
```

```
: """symmetric difference: set of elements in both set1 and set2
  #except those that are common in both."""

  #use ^ operator

  print(set1^set2)
```

```
{1, 2, 6, 7}
```

```
: #use symmetric_difference function
  print(set1.symmetric_difference(set2))
```

```
{1, 2, 6, 7}
```

```
: #find issubset()
  x = {"a","b","c","d","e"}
  y = {"c","d"}

  print("set 'x' is subset of 'y' ?", x.issubset(y)) #check x is subset of y

  #check y is subset of x
  print("set 'y' is subset of 'x' ?", y.issubset(x))
```

```
set 'x' is subset of 'y' ? False
set 'y' is subset of 'x' ? True
```

To perform the union operation, you have to use either the '**|**' operator or the **union()** method.
To perform the intersection operation, you have to use either the '**&**' operator or the **intersection()** method.
To perform the difference operation, you have to use either the '**-**' operator or **difference()** method.
To perform the symmetric difference operation, you have to use either the '^' operator or the **symmetric_difference()** method.
To check if a given set is a subset of another set, then you have to use the **issubset()** method.

Frozenset

Frozensets have all the characteristics of set data structure, but the only difference is that once if it is created, then we cannot modify the elements/values in it.
As tuples are the immutable lists, the frozensets are the immutable sets.
Frozensets can be created using the frozenset() method.

Since the sets are mutable, they are unhashable and cannot be used as the keys in a dictionary whereas the frozensets being immutable and hashable, can be used as the dictionary keys.

Frozenses supports methods like copy(), difference(), intersection(), isdisjoint(), issubset(), issuperset(), symmetric_difference() and union(). Being immutable it does not have methods that add or remove elements. Also similar to a set, the frozenset also does not support indexing.

Below are the examples that are discussed starting from the timestamp 13.30.

```
set1 = frozenset([1, 2, 3, 4])
set2 = frozenset([3, 4, 5, 6])

#try to add element into set1 gives an error
set1.add(5)
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-28-8f5ea3d0c7e1> in <module>()
      3
      4 #try to add element into set1 gives an error
----> 5 set1.add(5)

AttributeError: 'frozenset' object has no attribute 'add'
```

```
print(set1[1]) # frozen set doesn't support indexing
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-27-8fc108f08ec8> in <module>()
----> 1 print(set1[1]) # frozen set doesn't support indexing

TypeError: 'frozenset' object does not support indexing
```

```
print(set1 | set2) #union of 2 sets
```

```
frozenset({1, 2, 3, 4, 5, 6})
```

```
#intersection of two sets
print(set1 & set2)

#or
print(set1.intersection(set2))
```

```
frozenset({3, 4})
frozenset({3, 4})
```

```
#symmetric difference
print(set1 ^ set2)

#or
print(set1.symmetric_difference(set2))
```

```
frozenset({1, 2, 5, 6})
frozenset({1, 2, 5, 6})
```

5.5 Dictionary

- Dictionary is an unordered collection of items.
- In the other data structures discussed so far, we had only the values present in the form of elements/items whereas in a dictionary we have the items in the form of key:value pairs
- Similar to the set data structure, as the dictionary is an unordered data structure, we cannot access the elements using indexes.

Dictionary Creation

The below examples are discussed starting from the timestamp 1:25

```
#empty dictionary
my_dict = {}

#dictionary with integer keys
my_dict = {1: 'abc', 2: 'xyz'}
print(my_dict)

#dictionary with mixed keys
my_dict = {'name': 'satish', 1: ['abc', 'xyz']}
print(my_dict)


#create empty dictionary using dict()
my_dict = dict()

my_dict = dict([(1, 'abc'), (2, 'xyz')])    #create a dict with list of tuples
print(my_dict)
```

```
{1: 'abc', 2: 'xyz'}
{'name': 'satish', 1: ['abc', 'xyz']}
{1: 'abc', 2: 'xyz'}
```

- We can create an empty dictionary either using empty braces (ie., {}) or using the dict() function without passing any arguments.
- The items in a dictionary are stored in the form of key:value pairs. The keys of a dictionary should be hashable and no duplicate keys are allowed in a dictionary.
- We can have each key belonging to a different data type and this is supported by a dictionary.
- We can either separately define the key-value pairs inside a dictionary with the keys and values separated by a colon(:), or we can pass the key-value pairs either in the form of a tuple or a list.

➢ We also can use tuples and frozensets as the keys in a dictionary as they are hashable whereas lists and sets could not be used as the keys in a dictionary as they are not hashable.

Dictionary Access

Below examples are discussed starting from the timestamp 7:40 in the video.

```
my_dict = {'name': 'satish', 'age': 27, 'address': 'guntur'}

#get name
print(my_dict['name'])
```

```
satish
```

```
#if key is not present it gives KeyError
print(my_dict['degree'])
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-5-c5aba24e1656> in <module>()
      1 #if key is not present it gives KeyError
----> 2 print(my_dict['degree'])

KeyError: 'degree'
```

```
#another way of accessing key
print(my_dict.get('address'))
```

```
guntur
```

```
#if key is not present it will give None using get method
print(my_dict.get('degree'))
```

```
None
```

We can access the values from a dictionary with the help of their corresponding keys. The syntax for it would be **<dictionary-name>[<key-name>]**.

The same value can be accessed using the get() method of the dictionary class where we have to pass the **key** as an argument in the **get()** method.

If the specified key is not present in the dictionary, then it throws **KeyError**.

Add or Modify the dictionary items

The below examples are discussed starting from the timestamp 9:50 in the video.

```
my_dict = {'name': 'satish', 'age': 27, 'address': 'guntur'}

#update name
my_dict['name'] = 'raju'

print(my_dict)
```

```
{'name': 'raju', 'age': 27, 'address': 'guntur'}
```

```
#add new key
my_dict['degree'] = 'M.Tech'

print(my_dict)
```

```
{'name': 'raju', 'age': 27, 'address': 'guntur', 'degree': 'M.Tech'}
```

In the above example, if we want to modify the value of a key present in the dictionary, then it is as simple as assigning a value to the key.

my_dict[‘name’] = ‘raju’

In this example, it looks as if we are assigning the value ‘raju’ to the key ‘name’ in the dictionary.

In case, if we do not have the specified key in the dictionary, then it creates the key-value pair in the dictionary.

Deletion/Removal of elements from a dictionary

The below examples are discussed starting from the timestamp 11:20 in the video.

```
: #create a dictionary
my_dict = {'name': 'satish', 'age': 27, 'address': 'guntur'}

#remove a particular item
print(my_dict.pop('age'))

print(my_dict)
```

```
27
{'name': 'satish', 'address': 'guntur'}
```

```
: my_dict = {'name': 'satish', 'age': 27, 'address': 'guntur'}

#remove an arbitarty key
my_dict.popitem()

print(my_dict)
```

```
{'name': 'satish', 'age': 27}
```

```
: squares = {2: 4, 3: 9, 4: 16, 5: 25}

#delete particular key
del squares[2]

print(squares)
```

```
{3: 9, 4: 16, 5: 25}
```

```
: #remove all items
squares.clear()

print(squares)
```

```
{}
```

```
squares = {2: 4, 3: 9, 4: 16, 5: 25}

#delete dictionary itself
del squares

print(squares) #NameError because dict is deleted
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-16-355e8277492b> in <module>()
      4 del squares
      5
----> 6 print(squares) #NameError because dict is deleted

NameError: name 'squares' is not defined
```

- If we want to remove a key-value pair from the dictionary, then we have to use the **pop()** method and should pass the key as an argument in it. The specified key-value pair gets removed and the value associated with that key would be returned.
- The other way to use the same job is by using the **del** keyword. The syntax for it would be **del <dictionary_name>[<key>]**. This approach doesn't return any value.
- If we want to remove a random key-value pair, then we have to use the **popitem()** method of the dictionary class.
- If we want to remove all the key-value pairs, then we have to use the **clear()** method.
- If we want to delete the whole dictionary then the syntax would be **del <dictionary_name>**

Dictionary Methods

The below examples are discussed starting from the timestamp 13:35

```
squares = {2: 4, 3: 9, 4: 16, 5: 25}

my_dict = squares.copy()
print(my_dict)
```

```
{2: 4, 3: 9, 4: 16, 5: 25}
```

- We use the **copy()** method to return another copy of the given dictionary.

```
#fromkeys[seq[, v]] -> Return a new dictionary with keys from seq and value equal to v (defaults to None).
subjects = {}.fromkeys(['Math', 'English', 'Hindi'], 0)
print(subjects)
```

```
{'Math': 0, 'English': 0, 'Hindi': 0}
```

➢ We use the **fromkeys()** method by passing a sequence and a value as the arguments, in order to create a new dictionary with the elements in the sequence as the keys and the specified value, as the value for those keys.

```
subjects = {2:4, 3:9, 4:16, 5:25}
print(subjects.keys()) #return a new view of the dictionary keys
```

```
dict_keys([2, 3, 4, 5])
```

➢ We use the **keys()** method to return all the keys in the form of a list.

```
subjects = {2:4, 3:9, 4:16, 5:25}
print(subjects.values()) #return a new view of the dictionary values
```

```
dict_values([4, 9, 16, 25])
```

➢ We use the **values()** method to return all the values in the form of a list.

```
subjects = {2:4, 3:9, 4:16, 5:25}
print(subjects.items()) #return a new view of the dictionary items (key, value)
```

```
dict_items([(2, 4), (3, 9), (4, 16), (5, 25)])
```

➢ We use the **items()** method to return all the key-value pairs as a list of tuples.

```
#get list of all available methods and attributes of dictionary
d = {}
print(dir(d))
```

```
['__class__', '__contains__', '__delattr__', '__delitem__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__iter__', '__le__', '__len__', '__lt__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__setitem__', '__sizeof__', '__str__', '__subclasshook__', 'clear', 'copy', 'fromkeys', 'get', 'items', 'keys', 'pop', 'popitem', 'setdefault', 'update', 'values']
```

➢ We use the **dir()** method to display a list of all the available methods and attributes of a Dictionary.

Dictionary Comprehension

5.6 Strings

- ➢ A string is a sequence of characters.
- ➢ Computers do not deal with the characters. They only deal with the binary numbers. Even though we may see the characters on the screen, internally they are stored and are manipulated as 0's and 1's.
- ➢ The conversion of character to a number is called encoding and the reverse process is called decoding. ASCII and Unicode are the most popularly used encoding techniques.
- ➢ In Python, a string is a sequence of unicode characters.

How to create a string?

- ➢ Strings can be created by enclosing the characters either inside single quotes or double quotes or triple quotes.
- ➢ Mostly we use triple quotes only when we want to represent the multi-line strings or the docstrings.

Below example is discussed at the timestamp 0:40 in the video.

```
myString = 'Hello'

print(myString)


myString = "Hello"
print(myString)


myString = '''Hello'''
print(myString)
```

```
Hello
Hello
Hello
```

How to access the characters in a string?

- ➢ We can access the individual characters using the indexing and the range of characters using slicing.
- ➢ Indexing starts from **0** and ends with the **(total length of the string-1)**. Whenever we try to access any element with an index outside the above specified range, then it throws an error.
- ➢ We have to use only integers as the indexes to access the characters of a string. We cannot use float values or any other characters as the indexes.

➢ Python also allows negative indexing, but the traversal happens in the reverse order.
Below examples are discussed in the video starting from the timestamp 1:35 and 2:55 respectively.

```
: myString = "Hello"

#print first Character
print(myString[0])

#print last character using negative indexing
print(myString[-1])

#slicing 2nd to 5th character
print(myString[2:5])
```

```
H
o
llo
```

```
print(myString[15])
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-2-78353fed94bc> in <module>()
----> 1 print(myString[15])

IndexError: string index out of range
```

```
print(myString[1.5])
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-3-d32f87fd1591> in <module>()
----> 1 print(myString[1.5])

TypeError: string indices must be integers
```

How to change or delete a string?

➢ Strings are immutable objects. It means that once after a string is assigned with some values, it cannot be changed.

➢ Instead we can make reassignments to the same string variable but with a different value. Making changes to the existing values is not possible

Below example starts from the timestamp 3:50.

```
myString = "Hello"
myString[4] = 's' # strings are immutable
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-6-d63a28c2a378> in <module>()
      1 myString = "Hello"
----> 2 myString[4] = 's' # strings are immutable

TypeError: 'str' object does not support item assignment
```

We cannot delete or remove characters from a string. But deleting the string entirely is possible using the keyword del.

- ➢ We cannot modify elements/substrings of a given string, as strings are immutable objects.
- ➢ In order to delete a string, we use the **del** keyword followed by the string name.

The below example follows the above example in the video.

```
del myString # delete complete string
```

```
print(myString)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-8-60c083ddb862> in <module>()
----> 1 print(myString)

NameError: name 'myString' is not defined
```

String Operations

Concatenation

- ➢ Concatenation is the process of combining two or more strings into a single string.
- ➢ In order to combine the strings, we use the '+' operator between two string literals.
- ➢ In order to repeat the given string for a specified number of times, then we have to use '*' operator.

The next example was discussed starting from the timestamp 5:30 in the video.

```
s1 = "Hello "
s2 = "Satish"

#concatenation of 2 strings
print(s1 + s2)

#repeat string n times
print(s1 * 3)
```

```
Hello Satish
Hello Hello Hello
```

- ➢ As we are using the '+' operator, the strings 'Hello' and 'Satish' are getting concatenated.
- ➢ As we are using the '*' operator followed by the number 3, the string 'Hello' is occurring 3 times and all these are getting concatenated.

Iteration through String

As a string is a sequence of characters, we can iterate over the sequence each character-wise. The below example begins at the timestamp 6:20 in the video.

```
count = 0
for l in "Hello World":
    if l == 'o':
        count += 1
print(count, ' letters found')
```

```
2  letters found
```

In the above example, we are iterating over the string "Hello World" and in each iteration, the variable 'l' denotes each character from the string "Hello World".

String Membership Test

We can check if a character/substring is a member/part of the given string. We use the 'in' operator to check for this purpose. The below example was discussed at the timestamp 7:20 in the video.

```
print('l' in 'Hello World') #in operator to test membership
```

```
True
```

```
print('or' in 'Hello World')
```

```
True
```

In the first example, we are checking if the character ‘l’ is present in the string “Hello World”. As the character ‘l’ is present in the given string, it returns True.

In the second example, we are checking if the substring ‘or’ is present in the string “Hello World”. As the substring ‘or’ is present in the given string, it returns True.

String Methods

The below examples are discussed starting from the timestamp 7:50 in the video.

```
"Hello".lower()
```

```
'hello'
```

```
"Hello".upper()
```

```
'HELLO'
```

```
"This will split all words in a list".split()
```

```
['This', 'will', 'split', 'all', 'words', 'in', 'a', 'list']
```

```
' '.join(['This', 'will', 'split', 'all', 'words', 'in', 'a', 'list'])
```

```
'This will split all words in a list'
```

```
"Good Morning".find("Mo")
```

```
5
```

```
s1 = "Bad morning"

s2 = s1.replace("Bad", "Good")

print(s1)
print(s2)
```

```
Bad morning
Good morning
```

- We use the **lower()** function to convert the given string into lowercase.
- We use the **upper()** function to convert the given string into uppercase.
- We use the **split()** function to split the given string into a list of strings.
- We use the **join()** function to combine all the string elements in a given list into a single string format.
- We use the **find()** function to find out the index of a given character or substring in the given string.
- We can use the **replace()** function with 2 arguments, to replace an old substring with a new substring. But as strings are immutable objects, the replace() function doesn't perform any replacements on the original strings. It just returns a copy of the modified string.

Python Program to check if the given string is a palindrome

The below code snippet was explained in the video at the timestamp 12:20 in the video.

```
myStr = "Madam"

#convert entire string to either lower or upper
myStr = myStr.lower()

#reverse string
revStr = reversed(myStr)


#check if the string is equal to its reverse
if list(myStr) == list(revStr):
    print("Given String is palindrome")
else:
    print("Given String is not palindrome")
```

```
Given String is palindrome
```

- In the above example, we are first converting the given string into lower case (it is because the lower case and the uppercase of an alphabet are considered as two different alphabets)
- After converting the string into lowercase, we are reversing the string and then converting both the strings(the original one and the reverse form) into lists of characters(same in the order in which they appear in the strings) and are comparing each element corresponding to the same indexes.
- If all the characters are the same, then it returns True and we can declare the given string as a Palindrome.

Python Program to sort the words in the given string

Below example was discussed at the timestamp 14:00 in the video.

```
myStr = "python Program to Sort words in Alphabetic Order"

#breakdown the string into list of words
words = myStr.split()

#sort the list
words.sort()

#print Sorted words are
for word in words:
    print(word)
```

```
Alphabetic
Order
Program
Sort
in
python
to
words
```

- In the above example, we first have to split the given input string into a list of substrings.
- We then have to sort that list using the sort() function and then traverse through the sorted list and print the elements one by one. This sorting is done on the basis of alphabetical order.