Data Structures

Lists

A list is a container which holds comma-separated values (items or elements) between square brackets where items or elements need not all have the same type.
In general, we can define a list as an object that contains multiple data items (elements). The contents of a list can be changed during program execution. The size of a list can also change during execution, as elements are added or removed from it.
Note: There are much programming languages which allow us to create arrays, which are objects similar to lists. Lists serve the same purpose as arrays and have many more built-in capabilities. Traditional arrays can not be created in Python.

Examples of lists :

- numbers = [10, 20, 30, 40, 50]
- names = ["Sara", "David", "Warner", "Sandy"]
- student_info = ["Sara", 1, "Chemistry"]

operations on lists :

- Create a Python list
- List indices
- Add an item to the end of the list
- Insert an item at a given position
- Modify an element by using the index of the element
- Remove an item from the list
- Remove all items from the list
- Slice Elements from a List
- Remove the item at the given position in the list, and return it
- Return the index in the list of the first item whose value is x
- Return the number of times 'x' appear in the list
- Sort the items of the list in place
- Reverse the elements of the list in place
- Return a shallow copy of the list
- Search the Lists and find elements
- Lists are mutable
- Convert a list to a tuple in python?
- How to use the double colon [: :]?
- Find largest and the smallest items in a list
- Compare two lists in Python?
- Nested lists in Python
- How can I get the index of an element contained in the list?
- Using Lists as Stacks
- Using Lists as Queues
- Python List Exercises

Create a Python list

```
>>> my_list1 = [5, 12, 13, 14] # the list contains all integer values
>>> print(my_list1)
[5, 12, 13, 14]
>>> my_list2 = ['red', 'blue', 'black', 'white'] # the list contains all string
values
>>> print(my_list2)
['red', 'blue', 'black', 'white']
>>> my_list3 = ['red', 12, 112.12] # the list contains a string, an integer and
a float values
>>> print(my_list3)
['red', 12, 112.12]
>>>
```

A list without any element is called an empty list. See the following statements.

```
>>> my_list=[]
>>> print(my_list)
[]
>>>
```

Use + operator to create a new list that is a concatenation of two lists and use * operator to repeat a list. See the following statements.

```
>>> color_list1 = ["White", "Yellow"]
>>> color_list2 = ["Red", "Blue"]
>>> color_list3 = ["Green", "Black"]
>>> color_list = color_list1 + color_list2 + color_list3
>>> print(color_list)
['White', 'Yellow', 'Red', 'Blue', 'Green', 'Black']
>>> number = [1,2,3]
>>> print(number[0]*4)
4
>>> print(number*4)
[1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3]
>>>
```

List indices

List indices work the same way as string indices, list indices start at 0. If an index has a positive value it counts from the begriming and similarly it counts backward if the index has a negative value. As positive integers are used to index from the left end and negative integers are used to index from the right end, so every item of a list gives two alternatives indices. Let create a list called color_list with four items.

color_list=["RED", "Blue", "Green", "Black"]

Item	RED	Blue	Green	Black
Index (from left)	0	1	2	3
Index (from right)	-4	-3	-2	-1

If you give any index value which is out of range then interpreter creates an error message. See the following statements.

```
>>> color_list=["Red", "Blue", "Green", "Black"] # The list have four elements
indices start at 0 and end at 3
>>> color_list[0] # Return the First Element
'Red'
>>> print(color_list[0],color_list[3]) # Print First and Last Elements
Red Black
>>> color_list[-1] # Return Last Element
'Black'
>>> print(color_list[4]) # Creates Error as the indices is out of range
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
>>></module></stdin>
```

Add an item to the end of the list

```
>>> color_list=["Red", "Blue", "Green", "Black"]
>>> print(color_list)
['Red', 'Blue', 'Green', 'Black']
>>> color_list.append("Yellow")
>>> print(color_list)
['Red', 'Blue', 'Green', 'Black', 'Yellow']
>>>
```

Insert an item at a given position

```
>>> color_list=["Red", "Blue", "Green", "Black"]
>>> print(color_list)
['Red', 'Blue', 'Green', 'Black']
>>> color_list.insert(2, "White") #Insert an item at third position
>>> print(color_list)
['Red', 'Blue', 'White', 'Green', 'Black']
>>>
```

Modify an element by using the index of the element

```
>>> color_list=["Red", "Blue", "Green", "Black"]
>>> print(color_list)
['Red', 'Blue', 'Green', 'Black']
>>> color_list[2]="Yellow"  #Change the third color
>>> print(color_list)
['Red', 'Blue', 'Yellow', 'Black']
>>>
```

Remove an item from the list

```
>>> color_list=["Red", "Blue", "Green", "Black"]
>>> print(color_list)
['Red', 'Blue', 'Green', 'Black']
>>> color_list.remove("Black")
>>> print(color_list)
['Red', 'Blue', 'Green']
```

Remove all items from the list.

```
>>> color_list=["Red", "Blue", "Green", "Black"]
>>> print(color_list)
['Red', 'Blue', 'Green', 'Black']
>>> color_list.clear()
>>> print(color_list)
[]
>>>
```

List Slices

Lists can be sliced like strings and other sequences.

Syntax :

```
sliced_list = List_Name[startIndex:endIndex]
```

This refers to the items of a list starting at index startIndex and stopping just before index endIndex. The default values for list are 0 (startIndex) and the end (endIndex) of the list. If you omit both indices, the slice makes a copy of the original list. See the following statements.

```
>>> color_list=["Red", "Blue", "Green", "Black"] # The list have four elements
indices start at 0 and end at 3
>>> print(color_list[0:2]) # cut first two items
['Red', 'Blue']
>>> print(color_list[1:2]) # cut second item
['Blue']
>>> print(color_list[1:-2]) # cut second item
['Blue']
```

```
>>> print(color_list[:3]) # cut first three items
['Red', 'Blue', 'Green']
>>> print(color_list[:]) # Creates copy of original list
['Red', 'Blue', 'Green', 'Black']
>>>
```

Remove the item at the given position in the list, and return it

```
>>> color_list=["Red", "Blue", "Green", "Black"]
>>> print(color_list)
['Red', 'Blue', 'Green', 'Black']
>>> color_list.pop(2) # Remove second item and return it
'Green'
>>> print(color_list)
['Red', 'Blue', 'Black']
>>>
```

Return the index in the list of the first item whose value is x

```
>>> color_list=["Red", "Blue", "Green", "Black"]
>>> print(color_list)
['Red', 'Blue', 'Green', 'Black']
>>> color_list.index("Red")
0
>>> color_list.index("Black")
3
>>>
```

Return the number of times 'x' appear in the list.

```
>>> color_list=["Red", "Blue", "Green", "Black"]
>>> print(color_list)
['Red', 'Blue', 'Green', 'Black']
>>> color_list=["Red", "Blue", "Green", "Black", "Blue"]
>>> print(color_list)
['Red', 'Blue', 'Green', 'Black', 'Blue']
>>> color_list.count("Blue")
2
>>>
```

Sort the items of the list in place.

```
>>> color_list=["Red", "Blue", "Green", "Black"]
>>> print(color_list)
['Red', 'Blue', 'Green', 'Black']
>>> color_list.sort(key=None, reverse=False)
>>> print(color_list)
['Black', 'Blue', 'Green', 'Red']
>>>
```

Reverse the elements of the list in place

```
>>> color_list=["Red", "Blue", "Green", "Black"]
>>> print(color_list)
['Red', 'Blue', 'Green', 'Black']
>>> color_list.reverse()
```

```
>>> print(color_list)
['Black', 'Green', 'Blue', 'Red']
>>>
```

Return a shallow copy of the list.

```
>>> color_list=["Red", "Blue", "Green", "Black"]
>>> print(color_list)
['Red', 'Blue', 'Green', 'Black']
>>> color_list.copy()
['Red', 'Blue', 'Green', 'Black']
>>>
```

Search the Lists and find Elements

```
>>> color_list=["Red", "Blue", "Green", "Black"]
>>> print(color_list)
['Red', 'Blue', 'Green', 'Black']
>>> color_list.index("Green")
2
>>>
```

Lists are Mutable

Items in the list are mutable i.e. after creating a list you can change any item in the list. See the following statements.

```
>>> color_list=["Red", "Blue", "Green", "Black"]
>>> print(color_list[0])
Red
>>> color_list[0]="White" # Change the value of first item "Red" to "White"
>>> print(color_list)
['White', 'Blue', 'Green', 'Black']
>>> print(color_list[0])
White
>>>
```

Convert a list to a tuple in Python

```
>>> tupl=[1, 2, 3, 4]
>>> print(tupl)
[1, 2, 3, 4]
>>> listx=list(tupl)
>>> print(listx)
[1, 2, 3, 4]
>>>
```

How to use the double colon [: :]?

```
>>> listx=[1, 5, 7, 3, 2, 4, 6]
>>> print(listx)
[1, 5, 7, 3, 2, 4, 6]
>>> sublist=listx[2:7:2] #list[start:stop:step], #step specify an increment
between the elements to cut of the list.
>>> print(sublist)
[7, 2, 6]
>>> sublist=listx[::3] #returns a list with a jump every 2 times.
>>> print(sublist)
```

```
[1, 3, 6]
>>> sublist=listx[6:2:-2] #when step is negative the jump is made back
>>> print(sublist)
[6, 2]
>>>
```

Find the largest and the smallest item in a list

```
>>> listx=[5, 10, 3, 25, 7, 4, 15]
>>> print(listx)
[5, 10, 3, 25, 7, 4, 15]
>>> print(max(listx))  # the max() function of built-in allows to know the highest
value in the list.
25
>>> print(min(listx)) #the min() function of built-in allows to know the lowest
value in the list.
3
>>>
```

Compare two lists in Python

```
>>> listx1, listx2=[3, 5, 7, 9], [3, 5, 7, 9]
>>> print (listx1 == listx2)
True
>>> listx1, listx2=[9, 7, 5, 3], [3, 5, 7, 9]  #create two lists equal, but unsorted.
>>> print(listx1 == listx2)
False
>>> listx1, listx2 =[2, 3, 5, 7], [3, 5, 7, 9] #create two different list
>>> print(listx1 == listx2)
False
>>> print(listx1.sort() == listx2.sort())  #order and compare
True
>>>
```

Nested lists in Python

```
>>> listx = [["Hello", "World"], [0, 1, 2, 3, 4, 5]]
>>> print(listx)
[['Hello', 'World'], [0, 1, 2, 3, 4, 5]]
>>> listx = [["Hello", "World"], [0, 1, 2, 3, 3, 5]]
>>> print(listx)
[['Hello', 'World'], [0, 1, 2, 3, 3, 5]]
>>> print(listx[0][1])     #The first [] indicates the index of the outer list.
World
>>> print(listx[1][3])     #the second [] indicates the index nested lists.
3
>>> listx.append([True, False])        #add new items
>>> print(listx)
[['Hello', 'World'], [0, 1, 2, 3, 3, 5], [True, False]]
>>> listx[1][2]=4
>>> print(listx)
[['Hello', 'World'], [0, 1, 4, 3, 3, 5], [True, False]]     #update value items
>>>
```

How can I get the index of an element contained in the list?

```
>>> listy = list("HELLO WORLD")
>>> print(listy)
['H', 'E', 'L', 'L', 'O', ' ', 'W', 'O', 'R', 'L', 'D']
>>> index = listy.index("L")   #get index of the first item whose value is passed as par
ameter
>>> print(index)
2
>>> index = listy.index("L", 4)    #define the index from which you want to search
>>> print(index)
9
>>> index = listy.index("O", 3, 5) #define the segment of the list to be searched
>>> print(index)
4
>>>
```

Using Lists as Stacks

```
>>> color_list=["Red", "Blue", "Green", "Black"]
>>> print(color_list)
['Red', 'Blue', 'Green', 'Black']
>>> color_list.append("White")
>>> color_list.append("Yellow")
>>> print(color_list)
['Red', 'Blue', 'Green', 'Black', 'White', 'Yellow']
>>> color_list.pop()
'Yellow'
>>> color_list.pop()
'White'
>>> color_list.pop()
'Black'
>>> color_list
['Red', 'Blue', 'Green']
>>>
```

Using Lists as Queues

```
>>> from collections import deque
>>> color_list = deque(["Red", "Blue", "Green", "Black"])
>>> color_list.append("White")      # White arrive
>>> print(color_list)
deque(['Red', 'Blue', 'Green', 'Black', 'White'])
>>> color_list.append("Yellow")     # Yellow arrive
>>> print(color_list)
deque(['Red', 'Blue', 'Green', 'Black', 'White', 'Yellow'])
>>> color_list.popleft()            # The first to arrive now leaves
'Red'
>>> print(color_list)
deque(['Blue', 'Green', 'Black', 'White', 'Yellow'])
>>> color_list.popleft()            # The second to arrive now leaves
'Blue'
>>> print(color_list)
deque(['Green', 'Black', 'White', 'Yellow'])
>>> print(color_list)               # Remaining queue in order of arrival
deque(['Green', 'Black', 'White', 'Yellow'])
```

```
>>>
```

Tuples

A tuple is a container which holds a series of comma-separated values (items or elements) between parentheses such as an (x, y) co-ordinate. Tuples are like lists, except they are immutable (i.e. you cannot change its content once created) and can hold mix data types. Tuples play a sort of "struct" role in Python -- a convenient way to pass around a little logical, fixed size bundle of values.

Operations on tuple :

- Create a tuple
- How to get an item of the tuple in Python?
- How to know if an element exists within a tuple in Python?
- List to tuple
- Unpack a tuple in several variables
- Add item in Python tuple!
- Clone a tuple
- In Python how to know the number of times an item has repeated
- Remove an item from a tuple
- Slice a tuple
- How to get the index of an item of the tuple in Python?
- The size of a tuple
- How operators + and * are used with a Python tuple?
- Slice of a tuple using step parameter.
- Modify items of a tuple

Create a tuple?

To create a tuple, just list the values within parenthesis separated by commas. The "empty" tuple is just an empty pair of parenthesis

```
>>> #create an empty tuple
>>> tuplex = ()
>>> print (tuplex)
()
>>> #create a tuple with different data types
>>> tuplex = ('tuple', False, 3.2, 1)
>>> print (tuplex)
('tuple', False, 3.2, 1)
>>> #create a tuple with numbers, notation without parenthesis
>>> tuplex = 4, 7, 3, 8, 1
>>> print (tuplex)
(4, 7, 3, 8, 1)
>>> #create a tuple of one item, notation without parenthesis
>>> tuplex = 4,
>>> print (tuplex)
(4,)
>>> #create an empty tuple with tuple() function built-in Python
>>> tuplex = tuple()
>>> print (tuplex)
()
>>> #create a tuple from a iterable object
>>> tuplex = tuple([True, False])
>>> print (tuplex)
(True, False)
>>>
```

How to get an item of the tuple in Python?

```
>>> #create a tuple
>>> tuplex = ("w", 3, "r", "e", "s", "o", "u", "r", "c", "e")
>>> print(tuplex)
('w', 3, 'r', 'e', 's', 'o', 'u', 'r', 'c', 'e')
>>> #get item (4th element)of the tuple by index
>>> item = tuplex[3]
>>> print(item)
e
>>> #get item (4th element from last)by index negative
>>> item1 = tuplex[-4]
>>> print(item1)
u
>>>
```

How to know if an element exists within a tuple in Python?

```
>>> #create a tuple
>>> tuplex = ("w", 3, "r", "e", "s", "o", "u", "r", "c", "e")
>>> print(tuplex)
('w', 3, 'r', 'e', 's', 'o', 'u', 'r', 'c', 'e')
>>> #use in statement
>>> print("r" in tuplex)
True
>>> print(5 in tuplex)
False
>>>
```

List to tuple

```
>>> #create list
>>> listx = [5, 10, 7, 4, 15, 3]
>>> print(listx)
[5, 10, 7, 4, 15, 3]
>>> #use the tuple() function built-in Python, passing as parameter the list
>>> tuplex = tuple(listx)
>>> print(tuplex)
(5, 10, 7, 4, 15, 3)
>>>
```

Unpack a tuple in several variables

```
>>> #create a tuple
>>> tuplex = 4, 8, 3
>>> print(tuplex)
(4, 8, 3)
>>> n1, n2, n3 = tuplex
>>> #unpack a tuple in variables
>>> print(n1 + n2 + n3)
15
>>> #the number of variables must be equal to the number of items of the tuple
>>> n1, n2, n3, n4 = tuplex
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: need more than 3 values to unpack
>>>
```

Add item in Python tuple!

```
>>> #create a tuple
>>> tuplex = (4, 6, 2, 8, 3, 1)
>>> print(tuplex)
(4, 6, 2, 8, 3, 1)
>>> #tuples are immutable, so you can not add new elements
>>> #using merge of tuples with the + operator you can add an element and it will cr
eate a new tuple
>>> tuplex = tuplex + (9,)
>>> print(tuplex)
(4, 6, 2, 8, 3, 1, 9)
>>> #adding items in a specific index
>>> tuplex = tuplex[:5] + (15, 20, 25) + tuplex[:5]
>>> print(tuplex)
(4, 6, 2, 8, 3, 15, 20, 25, 4, 6, 2, 8, 3)
>>> #converting the tuple to list
>>> listx = list(tuplex)
>>> #use different ways to add items in list
>>> listx.append(30)
>>> tuplex = tuple(listx)
>>> print(tuplex)
(4, 6, 2, 8, 3, 15, 20, 25, 4, 6, 2, 8, 3, 30)
>>>
```

Clone a tuple

```
>>> from copy import deepcopy
>>> #create a tuple
>>> tuplex = ("HELLO", 5, [], True)
>>> print(tuplex)
('HELLO', 5, [], True)
>>> #make a copy of a tuple using deepcopy() function
>>> tuplex_clone = deepcopy(tuplex)
>>> tuplex_clone[2].append(50)
>>> print(tuplex_clone)
('HELLO', 5, [50], True)
>>> print(tuplex)
('HELLO', 5, [], True)
>>>
```

In Python how to know the number of times an item has repeated

```
>>> #create a tuple
>>> tuplex = 2, 4, 5, 6, 2, 3, 4, 4, 7
>>> print(tuplex)
(2, 4, 5, 6, 2, 3, 4, 4, 7)
>>> #return the number of times it appears in the tuple.
>>> count = tuplex.count(4)
>>> print(count)
3
>>> count = tuplex.count(7)
>>> print(count)
1
>>> count = tuplex.count(5)
>>> print (count)
1
```

```
15.>>>
```

Remove an item from a tuple

```
1.  >>> #create a tuple
2.  >>> tuplex = "w", 3, "d", "r", "e", "s", "l"
3.  >>> print(tuplex)
4.  ('w', 3, 'd', 'r', 'e', 's', 'l')
5.  >>> #tuples are immutable, so you can not remove elements
6.  >>> #using merge of tuples with the + operator you can remove an item and it will cr
    eate a new tuple
7.  >>> tuplex = tuplex[:2] + tuplex[3:]
8.  >>> print(tuplex)
9.  ('w', 3, 'r', 'e', 's', 'l')
10.>>> #converting the tuple to list
11.>>> listx = list(tuplex)
12.>>> #use different ways to remove an item of the list
13.>>> listx.remove("l")
14.>>> #converting the tuple to list
15.>>> tuplex = tuple(listx)
16.>>> print(tuplex)
17.('w', 3, 'r', 'e', 's')
18.>>>
```

Slice a tuple

```
1.  >>> #create a tuple
2.  >>> tuplex = (2, 4, 3, 5, 4, 6, 7, 8, 6, 1)
3.  >>> #used tuple[start:stop] the start index is inclusive and the stop index
4.  >>> _slice = tuplex[3:5]
5.  #is exclusive.
6.  >>> print(_slice)
7.  (5, 4)
8.  >>> #if the start index isn't defined, is taken from the beg inning of the tuple.
9.  >>> _slice = tuplex[:6]
10.>>> print(_slice)
11.(2, 4, 3, 5, 4, 6)
12.>>> #if the end index isn't defined, is taken until the end of the tuple
13.>>> _slice = tuplex[5:]
14.>>> print(_slice)
15.(6, 7, 8, 6, 1)
16.>>> #if neither is defined, returns the full tuple
17.>>> _slice = tuplex[:]
18.>>> print(_slice)
19.(2, 4, 3, 5, 4, 6, 7, 8, 6, 1)
20.>>> #The indexes can be defined with negative values
21.>>> _slice = tuplex[-8:-4]
22.>>> print(_slice)
23.(3, 5, 4, 6)
24.>>>
```

Find the index of an item of the tuple

```
1.  >>> #create a tuple
2.  >>> tuplex = tuple("index tuple")
3.  >>> print(tuplex)
4.  ('i', 'n', 'd', 'e', 'x', ' ', 't', 'u', 'p', 'l', 'e')
```

```
>>> #get index of the first item whose value is passed as parameter
>>> index = tuplex.index("p")
>>> print(index)
8
>>> #define the index from which you want to search
>>> index = tuplex.index("p", 5)
>>> print(index)
8
>>> #define the segment of the tuple to be searched
>>> index = tuplex.index("e", 3, 6)
>>> print(index)
3
>>> #if item not exists in the tuple return ValueError Exception
>>> index = tuplex.index("y")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: tuple.index(x): x not in tuple
>>>
```

The size of a tuple

```
>>> tuplex = tuple("w3resource")   #create a tuple
>>> print(tuplex)
('w', '3', 'r', 'e', 's', 'o', 'u', 'r', 'c', 'e')
>>> #use the len() function to known the lenght of tuple.
>>> print(len(tuplex))
10
>>>
```

How operators + and * are used with a Python tuple?

```
>>> #create a tuple
>>> tuplex = 5, #create a tuple
>>> #The * operator allow repeat the items in the tuple
>>> print(tuplex * 6)
(5, 5, 5, 5, 5, 5)
>>> #create a tuple with repeated items.
>>> tuplex = (5, 10, 15) * 4
>>> print(tuplex)
(5, 10, 15, 5, 10, 15, 5, 10, 15, 5, 10, 15)
>>> #create three tuples
>>> tuplex1 = (3, 6, 9, 12, 15)
>>> tuplex2 = ("w", 3, "r", "s", "o", "u", "r", "c", "e")
>>> tuplex3 = (True, False)
>>> #The + operator allow create a tuple joining two or more tuples
>>> tuplex = tuplex1 + tuplex2 + tuplex3
>>> print(tuplex)
(3, 6, 9, 12, 15, 'w', 3, 'r', 's', 'o', 'u', 'r', 'c', 'e', True, False)
>>>
```

Slice of a tuple using step parameter

```
>>> #create a tuple
>>> tuplex = tuple("HELLO WORLD")
>>> print(tuplex)
('H', 'E', 'L', 'L', 'O', ' ', 'W', 'O', 'R', 'L', 'D')
>>> #step specify an increment between the elements to cut of the tuple.
```

```
>>> _slice = tuplex[2:9:2] #tuple[start:stop:step]
>>> print(_slice)
('L', 'O', 'W', 'R')
>>> #returns a tuple with a jump every 3 items.
>>> _slice = tuplex[::4]
>>> print(_slice)
('H', 'O', 'R')
>>> #when step is negative the jump is made back
>>> _slice = tuplex[9:2:-4]
>>> print(_slice)
('L', ' ')
>>> #when step is negative the jump is made back
>>> _slice = tuplex[9:2:-3]
>>> print(_slice)
('L', 'W', 'L')
>>>
```

Modify items of a tuple

```
>>> #create a tuple
>>> tuplex = ("w", 3, "r", [], False)
>>> print(tuplex)
('w', 3, 'r', [], False)
>>> #tuples are immutable, so you can not modify items which are also immutable, a
s str, boolean, numbers etc.
>>> tuplex[3].append(200)
>>> print(tuplex)
('w', 3, 'r', [200], False)
>>>
```

Sets

A set object is an unordered collection of distinct hashable objects. It is commonly used in membership testing, removing duplicates from a sequence, and computing mathematical operations such as intersection, union, difference, and symmetric difference.
Sets support x in the set, len(set), and for x in set like other collections. Set is an unordered collection and does not record element position or order of insertion. Sets do not support indexing, slicing, or other sequence-like behavior.
There are currently two built-in set types, set, and frozenset. The set type is mutable — the contents can be changed using methods like add() and remove(). Since it is mutable, it has no hash value and cannot be used as either a dictionary key or as an element of another set. The frozenset type is immutable and hashable — its contents cannot be altered after it is created; it can, therefore, be used as a dictionary key or as an element of another set.

Operations on Sets

- Create a set in Python
- Iteration Over Sets
- Add member(s) in Python set
- Remove item(s) from Python set
- Intersection of sets
- Union of sets
- set difference in Python
- Symmetric difference
- issubset and issuperset
- Shallow copy of sets
- Clear sets

Create a set in Python

```
>>> #A new empty set
>>> setx = set()
>>> print(setx)
set()
>>> #A non empty set
>>> n = set([0, 1, 2, 3, 4, 5])
>>> print(n)
{0, 1, 2, 3, 4, 5}
>>>
```

Iteration Over Sets

We can move over each of the items in a set using a loop. However, since sets are unordered, it is undefined which order the iteration will follow.

```
>>> num_set = set([0, 1, 2, 3, 4, 5])
>>> for n in num_set:
  print(n)

 0
 1
 2
 3
 4
 5
>>>
```

Add member(s) in Python set

```
>>> #A new empty set
>>> color_set = set()
>>> #Add a single member
>>> color_set.add("Red")
>>> print(color_set)
{'Red'}
>>> #Add multiple items
>>> color_set.update(["Blue", "Green"])
>>> print(color_set)
{'Red', 'Blue', 'Green'}
>>>
```

Remove item(s) from Python set

pop(), remove() and discard() functions are used to remove individual item from a Python set. See the following examples :

pop() function :

```
>>> num_set = set([0, 1, 2, 3, 4, 5])
>>> num_set.pop()
0
>>> print(num_set)
{1, 2, 3, 4, 5}
>>> num_set.pop()
1
>>> print(num_set)
```

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

remove() function :

```
>>> num_set = set([0, 1, 2, 3, 4, 5])
>>> num_set.remove(0)
>>> print(num_set)
{1, 2, 3, 4, 5}
>>>
```

discard() function :

```
>>> num_set = set([0, 1, 2, 3, 4, 5])
>>> num_set.discard(3)
>>> print(num_set)
{0, 1, 2, 4, 5}
>>>
```

Intersection of sets

In mathematics, the intersection A ∩ B of two sets A and B is the set that contains all elements of A that also belong to B (or equivalently, all elements of B that also belong to A), but no other elements.

```
>>> #Intersection
>>> setx = set(["green", "blue"])
>>> sety = set(["blue", "yellow"])
>>> setz = setx & sety
>>> print(setz)
{'blue'}
>>>
```

Union of sets

In set theory, the union (denoted by ∪) of a collection of sets is the set of all distinct elements in the collection. It is one of the fundamental operations through which sets can be combined and related to each other.

```
>>> #Union
>>> setx = set(["green", "blue"])
>>> sety = set(["blue", "yellow"])
>>> seta = setx | sety
>>> print (seta)
{'yellow', 'blue', 'green'}
>>>
```

Set difference

```
>>> setx = set(["green", "blue"])
>>> sety = set(["blue", "yellow"])
>>> setz = setx & sety
>>> print(setz)
{'blue'}
>>> #Set difference
>>> setb = setx - setz
>>> print(setb)
{'green'}
```

```
10.>>>
```

Symmetric difference

```
1. >>> setx = set(["green", "blue"])
2. >>> sety = set(["blue", "yellow"])
3. >>> #Symmetric difference
4. >>> setc = setx ^ sety
5. >>> print(setc)
6. {'yellow', 'green'}
7. >>>
```

issubset and issuperset

```
1. >>> setx = set(["green", "blue"])
2. >>> sety = set(["blue", "yellow"])
3. >>> issubset = setx <= sety
4. >>> print(issubset)
5. False
6. >>> issuperset = setx >= sety
7. >>> print(issuperset)
8. False
9. >>>
```

Shallow copy of sets

```
1. >>> setx = set(["green", "blue"])
2. >>> sety = set(["blue", "green"])
3. >>> #A shallow copy
4. >>> setd = setx.copy()
5. >>> print(setd)
6. {'blue', 'green'}
7. >>>
```

Clear sets

```
1. >>> setx = set(["green", "blue"])
2. >>> #Clear AKA empty AKA erase
3. >>> sete = setx.copy()
4. >>> sete.clear()
5. >>> print(sete)
6. set()
7. >>>
```

Dictionary

Python dictionary is a container of the unordered set of objects like lists.The objects are surrounded by curly braces { }. The items in a dictionary are a comma-separated list of key:value pairs where keys and values are Python data type.
Each object or value accessed by key and keys are unique in the dictionary. As keys are used for indexing, they must be the immutable type (string, number, or tuple). You can create an empty dictionary using empty curly braces.

Operations on Dictionaries:

- Create a new dictionary in Python
- Get value by key in Python dictionary
- Add key/value to a dictionary in Python

- Iterate over Python dictionaries using for loops
- Remove a key from a Python dictionary
- Sort a Python dictionary by key
- Find the maximum and minimum value of a Python dictionary
- Concatenate two Python dictionaries into a new one
- Test whether a Python dictionary contains a specific key
- Find the length of a Python dictionary
- Python Dictionary - Exercises, Practice, Solution

Create a new dictionary in Python

```
>>> #Empty dictionary
>>> new_dict = dict()
>>> new_dict = {}
>>> print(new_dict)
{}
>>> #Dictionary with key-vlaue
>>> color = {"col1" : "Red", "col2" : "Green", "col3" : "Orange" }
>>> color
{'col2': 'Green', 'col3': 'Orange', 'col1': 'Red'}
>>>
```

Get value by key in Python dictionary

```
>>> #Declaring a dictionary
>>> dict = {1:20.5, 2:3.03, 3:23.22, 4:33.12}
>>> #Access value using key
>>> dict[1]
20.5
>>> dict[3]
23.22
>>> #Accessing value using get() method
>>> dict.get(1)
20.5
>>> dict.get(3)
23.22
>>>
```

Add key/value to a dictionary in Python

```
>>> #Declaring a dictionary with a single element
>>> dic = {'pdy1':'DICTIONARY'}
>>>print(dic)
{'pdy1': 'DICTIONARY'}
>>> dic['pdy2'] = 'STRING'
>>> print(dic)
{'pdy1': 'DICTIONARY', 'pdy2': 'STRING'}
>>>
>>> #Using update() method to add key-values pairs in to dictionary
>>> d = {0:10, 1:20}
>>> print(d)
{0: 10, 1: 20}
>>> d.update({2:30})
>>> print(d)
{0: 10, 1: 20, 2: 30}
>>>
```

Iterate over Python dictionaries using for loops

Code:

```
d = {'Red': 1, 'Green': 2, 'Blue': 3}
for color_key, value in d.items():
    print(color_key, 'corresponds to ', d[color_key])
```

Output:

```
>>>
Green corresponds to  2
Red corresponds to  1
Blue corresponds to  3
>>>
```

Remove a key from a Python dictionary

Code:

```
myDict = {'a':1,'b':2,'c':3,'d':4}
print(myDict)
if 'a' in myDict:
    del myDict['a']
print(myDict)
```

Output:

```
>>>
{'d': 4, 'a': 1, 'b': 2, 'c': 3}
{'d': 4, 'b': 2, 'c': 3}
>>>
```

Sort a Python dictionary by key

Code:

```
color_dict = {'red':'#FF0000',
          'green':'#008000',
          'black':'#000000',
          'white':'#FFFFFF'}

for key in sorted(color_dict):
    print("%s: %s" % (key, color_dict[key]))
```

Output:

```
>>>
black: #000000
green: #008000
red: #FF0000
white: #FFFFFF
>>>
```

Find the maximum and minimum value of a Python dictionary

Code:

```
my_dict = {'x':500, 'y':5874, 'z': 560}

key_max = max(my_dict.keys(), key=(lambda k: my_dict[k]))
key_min = min(my_dict.keys(), key=(lambda k: my_dict[k]))

print('Maximum Value: ',my_dict[key_max])
print('Minimum Value: ',my_dict[key_min])
```

Output:

```
>>>
```

```
Maximum Value:  5874
Minimum Value:  500
>>>
```

Concatenate two Python dictionaries into a new one

Code:

```
dic1={1:10, 2:20}
dic2={3:30, 4:40}
dic3={5:50,6:60}
dic4 = {}
for d in (dic1, dic2, dic3): dic4.update(d)
print(dic4)
```

Output:

```
>>>
{1: 10, 2: 20, 3: 30, 4: 40, 5: 50, 6: 60}
>>>
```

Test whether a Python dictionary contains a specific key

Code:

```
fruits = {}
fruits["apple"] = 1
fruits["mango"] = 2
fruits["banana"] = 4

# Use in.
if "mango" in fruits:
    print("Has mango")
else:
    print("No mango")

# Use in on nonexistent key.
if "orange" in fruits:
    print("Has orange")
else:
    print("No orange")
```

Output

```
>>>
Has mango
No orange
>>>
```

Find the length of a Python dictionary

Code:

```
fruits = {"mango": 2, "orange": 6}

# Use len() function <strong>to get the length of the dictionary</strong>
print("Length:", len(fruits))
```

Output:

```
>>>
Length: 2
```

Sequences:

The most basic data structure in Python is the **sequence**.

Sequence Types -- str, unicode, list, tuple, buffer, xrange

There are six sequence types: strings, Unicode strings, lists, tuples, buffers, and xrange objects.

Operation	Result	Notes
x in *s*	True if an item of *s* is equal to *x*, else False	(1)
x not in *s*	False if an item of *s* is equal to *x*, else True	(1)
s + *t*	the concatenation of *s* and *t*	(6)
s * *n* , *n* * *s*	*n* shallow copies of *s* concatenated	(2)
s[*i*]	*i*'th item of *s*, origin 0	(3)
s[*i*:*j*]	slice of *s* from *i* to *j*	(3), (4)
s[*i*:*j*:*k*]	slice of *s* from *i* to *j* with step *k*	(3), (5)
len(*s*)	length of *s*	
min(*s*)	smallest item of *s*	
max(*s*)	largest item of *s*	

Notes:

(1)

When *s* is a string or Unicode string object the in and not in operations act like a substring test. In Python versions before 2.3, *x* had to be a string of length 1. In Python 2.3 and beyond, *x* may be a string of any length.

(2)

Values of *n* less than 0 are treated as 0 (which yields an empty sequence of the same type as *s*). Note also that the copies are shallow; nested structures are not copied. This often haunts new Python programmers; consider:

```
>>> lists = [[]] * 3
>>> lists
[[], [], []]
>>> lists[0].append(3)
>>> lists
[[3], [3], [3]]
```

What has happened is that [[]] is a one-element list containing an empty list, so all three elements of [[]] * 3 are (pointers to) this single empty list. Modifying any of the elements of listsmodifies this single list. You can create a list of different lists this way:

```
>>> lists = [[] for i in range(3)]
>>> lists[0].append(3)
>>> lists[1].append(5)
>>> lists[2].append(7)
>>> lists
[[3], [5], [7]]
```

(3)

If *i* or *j* is negative, the index is relative to the end of the string: len(*s*) + *i* or len(*s*) + *j* is substituted. But note that -0 is still 0.

(4)

The slice of *s* from *i* to *j* is defined as the sequence of items with index *k* such that *i* <= *k* < *j*. If *i* or *j* is greater than len(*s*), use len(*s*). If *i* is omitted or None, use 0. If *j* is omitted or None, use len(*s*). If *i* is greater than or equal to *j*, the slice is empty.

(5)

The slice of *s* from *i* to *j* with step *k* is defined as the sequence of items with index $x = i + n*k$ such that $0 \leq n < \frac{j-i}{k}$. In other words, the indices are i, i+k, i+2*k, i+3*k and so on, stopping when *j* is reached (but never including *j*). If *i* or *j* is greater than len(*s*), use len(*s*). If *i* or *j* are omitted or None, they become ``end'' values (which end depends on the sign of *k*). Note, *k* cannot be zero. If *k* is None, it is treated like 1.

(6)

If *s* and *t* are both strings, some Python implementations such as CPython can usually perform an in-place optimization for assignments of the form *s=s+t* or *s+=t*. When applicable, this optimization makes quadratic run-time much less likely. This optimization is both version and implementation dependent. For performance sensitive code, it is preferable to use the str.join() method which assures consistent linear concatenation performance across versions and implementations. Changed in version 2.4: Formerly, string concatenation never occurred in-place.

Comprehensions

Comprehensions are constructs that allow sequences to be built from other sequences. Python 2.0 introduced list comprehensions and Python 3.0 comes with dictionary and set comprehensions.

List Comprehensions

A list comprehension consists of the following parts:

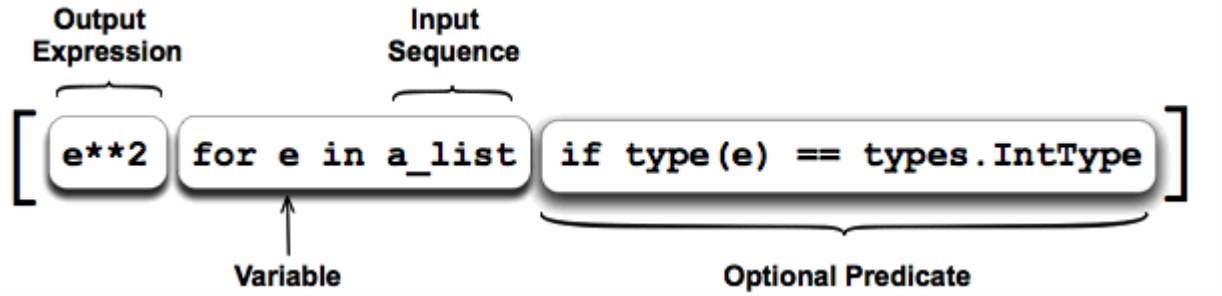
- An Input Sequence.
- A Variable representing members of the input sequence.
- An Optional Predicate expression.
- An Output Expression producing elements of the output list from members of the Input Sequence that satisfy the predicate.

Say we need to obtain a list of all the integers in a sequence and then square them:

```
a_list = [1, '4', 9, 'a', 0, 4]

squared_ints = [ e**2 for e in a_list if type(e) == types.IntType ]

print squared_ints
# [ 1, 81, 0, 16 ]
```



- The iterator part iterates through each member **e** of the input sequence **a_list**.
- The predicate checks if the member is an integer.
- If the member is an integer then it is passed to the output expression, squared, to become a member of the output list.

Much the same results can be achieved using the built in functions, **map**, **filter** and the anonymous **lambda** function.

The filter function applies a predicate to a sequence:

```
filter(lambda e: type(e) == types.IntType, a_list)
```

Map modifies each member of a sequence:

```
map(lambda e: e**2, a_list)
```

The two can be combined:

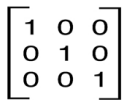
```
map(lambda e: e**2, filter(lambda e: type(e) == types.IntType, a_list))
```

The above example involves function calls to **map**, **filter**, **type** and two calls to **lambda**. Function calls in Python are expensive. Furthermore the input sequence is traversed through twice and an intermediate list is produced by filter.

The list comprehension is enclosed within a list so, it is immediately evident that a list is being produced. There is only one function call to **type** and no call to the cryptic **lambda** instead the list comprehension uses a conventional iterator, an expression and an if expression for the optional predicate.

Nested Comprehensions

An identity matrix of size n is an n by n square matrix with ones on the main diagonal and zeros elsewhere. A 3 by 3 identity matrix is:



In python we can represent such a matrix by a list of lists, where each sub-list represents a row. A 3 by 3 matrix would be represented by the following list:

```
[ [ 1, 0, 0 ],
  [ 0, 1, 0 ],
  [ 0, 0, 1 ] ]
```

The above matrix can be generated by the following comprehension:

```
[ [ 1 if item_idx == row_idx else 0 for item_idx in range(0, 3) ] for row_idx in range(0, 3) ]
```

Techniques

Using `zip()` and dealing with two or more elements at a time:

```
['%s=%s' % (n, v) for n, v in zip(self.all_names, self)]
```

Multiple types (auto unpacking of a tuple):

```
[f(v) for (n, f), v in zip(cls.all_slots, values)]
```

A two-level list comprehension using `os.walk()`:

```
# Comprehensions/os_walk_comprehension.py
import os
restFiles = [os.path.join(d[0], f) for d in os.walk(".")
             for f in d[2] if f.endswith(".rst")]
for r in restFiles:
    print(r)
```

Set Comprehensions

Set comprehensions allow sets to be constructed using the same principles as list comprehensions, the only difference is that resulting sequence is a set.

Say we have a list of names. The list can contain names which only differ in the case used to represent them, duplicates and names consisting of only one character. We are only interested in names longer then one character and wish to represent all names in the same format: The first letter should be capitalised, all other characters should be lower case.

Given the list:

```
names = [ 'Bob', 'JOHN', 'alice', 'bob', 'ALICE', 'J', 'Bob' ]
```

We require the set:

```
{ 'Bob', 'John', 'Alice' }
```

Note the new syntax for denoting a set. Members are enclosed in curly braces.

The following set comprehension accomplishes this:

```
{ name[0].upper() + name[1:].lower() for name in names if len(name) > 1 }
```

Dictionary Comprehensions

Say we have a dictionary the keys of which are characters and the values of which map to the number of times that character appears in some text. The dictionary currently distinguishes between upper and lower case characters.

We require a dictionary in which the occurrences of upper and lower case characters are combined:

```
mcase = {'a':10, 'b': 34, 'A': 7, 'Z':3}

mcase_frequency = { k.lower() : mcase.get(k.lower(), 0) + mcase.get(k.upper(), 0) for k in
mcase.keys() }

# mcase_frequency == {'a': 17, 'z': 3, 'b': 34}
```