Python – Lists

List is one of the built-in data types in Python. A Python list is a sequence of comma separated items, enclosed in square brackets [ ]. The items in a Python list need not be of the same data type.

Following are some examples of Python lists −

```
list1 = ["Rohan", "Physics", 21, 69.75]
list2 = [1, 2, 3, 4, 5]
list3 = ["a", "b", "c", "d"]
list4 = [25.50, True, -55, 1+2j]
```

In Python, a list is a sequence data type. It is an ordered collection of items. Each item in a list has a unique position index, starting from 0.

A list in Python is similar to an array in C, C++ or Java. However, the major difference is that in C/C++/Java, the array elements must be of same type. On the other hand, Python lists may have objects of different data types.

A Python list is mutable. Any item from the list can be accessed using its index, and can be modified. One or more objects from the list can be removed or added. A list may have same item at more than one index positions.

Python List Operations

In Python, List is a sequence. Hence, we can concatenate two lists with "+" operator and concatenate multiple copies of a list with "*" operator. The membership operators "in" and "not in" work with list object.

Python Expression	Results	Description
[1, 2, 3] + [4, 5, 6]	[1, 2, 3, 4, 5, 6]	Concatenation
['Hi!'] * 4	['Hi!', 'Hi!', 'Hi!', 'Hi!']	Repetition
3 in [1, 2, 3]	True	Membership

Python - Access List Items

In Python, a list is a sequence. Each object in the list is accessible with its index. The index starts from 0. Index or the last item in the list is "length-1". To access the values in a list, use the square brackets for slicing along with the index or indices to obtain value available at that index.

The slice operator fetches one or more items from the list. Put index on square brackets to retrieve item at its position.

obj = list1[i]

Example 1

Take a look at the following example −

```
list1 = ["Rohan", "Physics", 21, 69.75]
list2 = [1, 2, 3, 4, 5]

print ("Item at 0th index in list1: ", list1[0])
print ("Item at index 2 in list2: ", list2[2])
```

It will produce the following **output** −

```
Item at 0th index in list1: Rohan
Item at index 2 in list2: 3
```

Python allows negative index to be used with any sequence type. The "-1" index refers to the last item in the list.

Example 2

Let's take another example −

```
list1 = ["a", "b", "c", "d"]
list2 = [25.50, True, -55, 1+2j]

print ("Item at 0th index in list1: ", list1[-1])
print ("Item at index 2 in list2: ", list2[-3])
```

It will produce the following **output** −

```
Item at 0th index in list1: d
Item at index 2 in list2: True
```

The slice operator extracts a sublist from the original list.

Sublist = list1[i:j]

Parameters

- **i** − index of the first item in the sublist
- **j** − index of the item next to the last in the sublist

This will return a slice from ith to (j-1)th items from the **list1**.

Example 3

While slicing, both operands "i" and "j" are optional. If not used, "i" is 0 and "j" is the last item in the list. Negative index can be used in slicing. Take a look at the following example −

```
list1 = ["a", "b", "c", "d"]
list2 = [25.50, True, -55, 1+2j]

print ("Items from index 1 to 2 in list1: ", list1[1:3])
print ("Items from index 0 to 1 in list2: ", list2[0:2])
```

It will produce the following **output** −

```
Items from index 1 to 2 in list1: ['b', 'c']
Items from index 0 to 1 in list2: [25.5, True]
```

Example 4

```
list1 = ["a", "b", "c", "d"]
list2 = [25.50, True, -55, 1+2j]
list4 = ["Rohan", "Physics", 21, 69.75]
list3 = [1, 2, 3, 4, 5]

print ("Items from index 1 to last in list1: ", list1[1:])
print ("Items from index 0 to 1 in list2: ", list2[:2])
print ("Items from index 2 to last in list3", list3[2:-1])
print ("Items from index 0 to index last in list4", list4[:])
```

It will produce the following **output** −

```
Items from index 1 to last in list1: ['b', 'c', 'd']
Items from index 0 to 1 in list2: [25.5, True]
Items from index 2 to last in list3 [3, 4]
Items from index 0 to index last in list4 ['Rohan', 'Physics', 21, 69.75]
```

Python - Change List Items

List is a mutable data type in Python. It means, the contents of list can be modified in place, after the object is stored in the memory. You can assign a new value at a given index position in the list

Syntax

```
list1[i] = newvalue
```

Example 1

In the following code, we change the value at index 2 of the given list.

```
list3 = [1, 2, 3, 4, 5]
print ("Original list ", list3)
list3[2] = 10
print ("List after changing value at index 2: ", list3)
```

It will produce the following **output** −

```
Original list [1, 2, 3, 4, 5]
List after changing value at index 2: [1, 2, 10, 4, 5]
```

You can replace more consecutive items in a list with another sublist.

Example 2

In the following code, items at index 1 and 2 are replaced by items in another sublist.

```
list1 = ["a", "b", "c", "d"]

print ("Original list: ", list1)

list2 = ['Y', 'Z']
list1[1:3] = list2

print ("List after changing with sublist: ", list1)
```

It will produce the following **output** −

```
Original list: ['a', 'b', 'c', 'd']
List after changing with sublist: ['a', 'Y', 'Z', 'd']
```

Example 3

If the source sublist has more items than the slice to be replaced, the extra items in the source will be inserted. Take a look at the following code −

```
list1 = ["a", "b", "c", "d"]
print ("Original list: ", list1)
list2 = ['X','Y', 'Z']
list1[1:3] = list2
print ("List after changing with sublist: ", list1)
```

It will produce the following **output** −

```
Original list: ['a', 'b', 'c', 'd']
List after changing with sublist: ['a', 'X', 'Y', 'Z', 'd']
```

Example 4

If the sublist with which a slice of original list is to be replaced, has lesser items, the items with match will be replaced and rest of the items in original list will be removed.

In the following code, we try to replace "b" and "c" with "Z" (one less item than items to be replaced). It results in Z replacing b and c removed.

```
list1 = ["a", "b", "c", "d"]
print ("Original list: ", list1)
list2 = ['Z']
list1[1:3] = list2
print ("List after changing with sublist: ", list1)
```

It will produce the following **output** −

```
Original list: ['a', 'b', 'c', 'd']
List after changing with sublist: ['a', 'Z', 'd']
```

Python - Add List Items

There are two methods of the **list** class, append() and insert(), that are used to add items to an existing list.

Example 1

The **append()** method adds the item at the end of an existing list.

```
list1 = ["a", "b", "c", "d"]
print ("Original list: ", list1)
list1.append('e')
print ("List after appending: ", list1)
```

Output

```
Original list: ['a', 'b', 'c', 'd']
List after appending: ['a', 'b', 'c', 'd', 'e']
```

Example 2

The **insert()** method inserts the item at a specified index in the list.

```
list1 = ["Rohan", "Physics", 21, 69.75]
print ("Original list ", list1)

list1.insert(2, 'Chemistry')
print ("List after appending: ", list1)

list1.insert(-1, 'Pass')
print ("List after appending: ", list1)
```

Output

```
Original list ['Rohan', 'Physics', 21, 69.75]
List after appending: ['Rohan', 'Physics', 'Chemistry', 21, 69.75]
List after appending: ['Rohan', 'Physics', 'Chemistry', 21, 'Pass', 69.75]
```

We know that "-1" index points to the last item in the list. However, note that, the item at index "-1" in the original list is 69.75. This index is not refreshed after appending 'chemistry'. Hence, 'Pass' is not inserted at the updated index "-1", but the previous index "-1".

Python - Remove List Items

The list class methods **remove()** and **pop()** both can remove an item from a list. The difference between them is that remove() removes the object given as argument, while pop() removes an item at the given index.

Using the remove() Method

The following example shows how you can use the remove() method to remove list items −

```
list1 = ["Rohan", "Physics", 21, 69.75]
print ("Original list: ", list1)

list1.remove("Physics")
print ("List after removing: ", list1)
```

It will produce the following **output** −

```
Original list: ['Rohan', 'Physics', 21, 69.75]
List after removing: ['Rohan', 21, 69.75]
```

Using the pop() Method

The following example shows how you can use the pop() method to remove list items −

```
list2 = [25.50, True, -55, 1+2j]
print ("Original list: ", list2)
list2.pop(2)
print ("List after popping: ", list2)
```

It will produce the following **output** −

```
Original list: [25.5, True, -55, (1+2j)]
List after popping: [25.5, True, (1+2j)]
```

Using the "del" Keyword

Python has the "del" keyword that deletes any Python object from the memory.

Example

We can use "del" to delete an item from a list. Take a look at the following example −

```
list1 = ["a", "b", "c", "d"]
print ("Original list: ", list1)
del list1[2]
print ("List after deleting: ", list1)
```

It will produce the following **output** −

```
Original list: ['a', 'b', 'c', 'd']
List after deleting: ['a', 'b', 'd']
```

Example

You can delete a series of consecutive items from a list with the slicing operator. Take a look at the following example −

```
list2 = [25.50, True, -55, 1+2j]
print ("List before deleting: ", list2)
del list2[0:2]
print ("List after deleting: ", list2)
```

It will produce the following **output** −

```
List before deleting: [25.5, True, -55, (1+2j)]
List after deleting: [-55, (1+2j)]
```

Python - Loop Lists

You can traverse the items in a list with Python's **for** loop construct. The traversal can be done, using list as an iterator or with the help of index.

Syntax

Python list gives an iterator object. To iterate a list, use the for statement as follows −

```
for obj in list:
   . . .
   . . .
```

Example 1

Take a look at the following example −

```
lst = [25, 12, 10, -21, 10, 100]
for num in lst:
  print (num, end = ' ')
```

Output

```
25 12 10 -21 10 100
```

Example 2

To iterate through the items in a list, obtain the range object of integers "0" to "len-1". See the following example −

```
lst = [25, 12, 10, -21, 10, 100]
indices = range(len(lst))
for i in indices:
  print ("lst[{}]: ".format(i), lst[i])
```

Output

```
lst[0]: 25
lst[1]: 12
lst[2]: 10
lst[3]: -21
lst[4]: 10
lst[5]: 100
```

Python - List Comprehension

List comprehension is a very powerful programming tool. It is similar to set builder notation in mathematics. It is a concise way to create new list by performing some kind of process on each item on existing list. List comprehension is considerably faster than processing a list by for loop.

Example 1

Suppose we want to separate each letter in a string and put all non-vowel letters in a list object. We can do it by a **for** loop as shown below −

```
chars=[]
for ch in 'TutorialsPoint':
  if ch not in 'aeiou':
    chars.append(ch)
```

```
print (chars)
```

The chars list object is displayed as follows −

```
['T', 't', 'r', 'l', 's', 'P', 'n', 't']
```

List Comprehension Technique

We can easily get the same result by a list comprehension technique. A general usage of list comprehension is as follows −

```
listObj = [x for x in iterable]
```

Applying this, chars list can be constructed by the following statement −

```
chars = [ char for char in 'TutorialsPoint' if char not in 'aeiou']
print (chars)
```

The chars list will be displayed as before −

```
['T', 't', 'r', 'l', 's', 'P', 'n', 't']
```

Example 2

The following example uses list comprehension to build a list of squares of numbers between 1 to 10

```
squares = [x*x for x in range(1,11)]
print (squares)
```

The squares list object is −

```
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Nested Loops in List Comprehension

In the following example, all combinations of items from two lists in the form of a tuple are added in a third list object.

Example 3

```
list1=[1,2,3]
```

```
list2=[4,5,6]
CombLst=[(x,y) for x in list1 for y in list2]
print (CombLst)
```

It will produce the following **output** −

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

Condition in List Comprehension

The following statement will create a list of all even numbers between 1 to 20.

Example 4

```
list1=[x for x in range(1,21) if x%2==0]
print (list1)
```

It will produce the following **output** −

```
[2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
```

Python - Sort Lists

The sort() method of list class rearranges the items in ascending or descending order with the use of lexicographical ordering mechanism. The sorting is in-place, in the sense the rearrangement takes place in the same list object, and that it doesn't return a new object.

Syntax

```
list1.sort(key, reverse)
```

Parameters

- **Key** − The function applied to each item in the list. The return value is used to perform sort. Optional
- **reverse** − Boolean value. If set to True, the sort takes place in descending order. Optional

Return value

This method returns None.

Example 1

Now let's take a look at some examples to understand how we can sort lists in Python −

```
list1 = ['physics', 'Biology', 'chemistry', 'maths']
print ("list before sort", list1)
list1.sort()
print ("list after sort : ", list1)
print ("Descending sort")

list2 = [10,16, 9, 24, 5]
print ("list before sort", list2)
list2.sort()
print ("list after sort : ", list2)
```

It will produce the following **output** −

```
list before sort ['physics', 'Biology', 'chemistry', 'maths']
list after sort: ['Biology', 'chemistry', 'maths', 'physics']
Descending sort
list before sort [10, 16, 9, 24, 5]
list after sort : [5, 9, 10, 16, 24]
```

Example 2

In this example, the str.lower() method is used as key parameter in sort() method.

```
list1 = ['Physics', 'biology', 'Biomechanics', 'psychology']
print ("list before sort", list1)
list1.sort(key=str.lower)
print ("list after sort : ", list1)
```

It will produce the following **output** −

```
list before sort ['Physics', 'biology', 'Biomechanics', 'psychology']
list after sort : ['biology', 'Biomechanics', 'Physics', 'psychology']
```

Example 3

Let us use a user-defined function as the key parameter in sort() method. The myfunction() uses % operator to return the remainder, based on which the sort is done.

```
def myfunction(x):
   return x%10
list1 = [17, 23, 46, 51, 90]
print ("list before sort", list1)
list1.sort(key=myfunction)
print ("list after sort : ", list1)
```

It will produce the following **output** −

```
list before sort [17, 23, 46, 51, 90]
list after sort: [90, 51, 23, 46, 17]
```

Python - Copy Lists

In Python, a variable is just a label or reference to the object in the memory. Hence, the assignment "lst1 = lst" refers to the same list object in the memory. Take a look at the following example −

```
lst = [10, 20]
print ("lst:", lst, "id(lst):",id(lst))
lst1 = lst
print ("lst1:", lst1, "id(lst1):",id(lst1))
```

It will produce the following **output** −

```
lst: [10, 20] id(lst): 1677677188288
lst1: [10, 20] id(lst1): 1677677188288
```

As a result, if we update "lst", it will automatically reflect in "lst1". Change lst[0] to 100

```
lst[0]=100
print ("lst:", lst, "id(lst):",id(lst))
print ("lst1:", lst1, "id(lst1):",id(lst1))
```

It will produce the following **output** −

```
lst: [100, 20] id(lst): 1677677188288
lst1: [100, 20] id(lst1): 1677677188288
```

Hence, we can say that "lst1" is not the physical copy of "lst".

Using the Copy Method of List Class

Python's list class has a copy() method to create a new physical copy of a list object.

Syntax

```
lst1 = lst.copy()
```

The new list object will have a different id() value. The following example demonstrates this –

```
lst = [10, 20]
lst1 = lst.copy()
print ("lst:", lst, "id(lst):",id(lst))

print ("lst1:", lst1, "id(lst1):",id(lst1))
```

It will produce the following **output** −
lst: [10, 20] id(lst): 1677678705472
lst1: [10, 20] id(lst1): 1677678706304

Even if the two lists have same data, they have different id() value, hence they are two different objects and "lst1" is a copy of "lst".

If we try to modify "lst", it will not reflect in "lst1". See the following example −

```
lst[0]=100
print ("lst:", lst, "id(lst):",id(lst))
print ("lst1:", lst1, "id(lst1):",id(lst1))
```

It will produce the following **output** −
lst: [100, 20] id(lst): 1677678705472
lst1: [10, 20] id(lst1): 1677678706304

Python - Join Lists

In Python, List is classified as a sequence type object. It is a collection of items, which may be of different data types, with each item having a positional index starting with 0. You can use different ways to join two Python lists.

All the sequence type objects support concatenation operator, with which two lists can be joined.

```
L1 = [10,20,30,40]
L2 = ['one', 'two', 'three', 'four']
L3 = L1+L2

print ("Joined list:", L3)
```

It will produce the following **output** −
Joined list: [10, 20, 30, 40, 'one', 'two', 'three', 'four']

You can also use the augmented concatenation operator with "+=" symbol to append L2 to L1

```
L1 = [10,20,30,40]
L2 = ['one', 'two', 'three', 'four']
L1+=L2
print ("Joined list:", L1)
```

The same result can be obtained by using the extend() method. Here, we need to extend L1 so as to add elements from L2 in it.

```
L1 = [10,20,30,40]
```

```
L2 = ['one', 'two', 'three', 'four']
L1.extend(L2)

print ("Joined list:", L1)
```

To add items from one list to another, a classical iterative solution also works. Traverse items of second list with a for loop, and append each item in the first.

```
L1 = [10,20,30,40]
L2 = ['one', 'two', 'three', 'four']

for x in L2:
  L1.append(x)

print ("Joined list:", L1)
```

A slightly complex approach for merging two lists is using list comprehension, as following code shows −

```
L1 = [10,20,30,40]
L2 = ['one', 'two', 'three', 'four']
L3 = [y for x in [L1, L2] for y in x]
print ("Joined list:", L3)
```

Python - List Methods

Python's **list** class includes the following methods using which you can add, update, and delete list items −

Sr.No	**Methods & Description**
1	**list.append(obj)** Appends object obj to list
2	**list.clear()** Clears the contents of list
3	**list.copy()** Returns a copy of the list object
4	**list.count(obj)** Returns count of how many times obj occurs in list
5	**list.extend(seq)** Appends the contents of seq to list

6	**list.index(obj)** Returns the lowest index in list that obj appears
7	**list.insert(index, obj)** Inserts object obj into list at offset index
8	**list.pop(obj=list[-1])** Removes and returns last object or obj from list
9	**list.remove(obj)** Removes object obj from list
10	**list.reverse()** Reverses objects of list in place
11	**list.sort([func])** Sorts objects of list, use compare func if given

Assignment

1) Python program to find unique numbers in a given list.
2) Python program to find sum of all numbers in a list.
3) Python program to create a list of 5 random integers.
4) Python program to sort a list of strings on the number of alphabets in each word.
5) Python program to create a list of integers representing each character in a string.

Python – Tuples

Tuple is one of the built-in data types in Python. A Python tuple is a sequence of comma separated items, enclosed in parentheses (). The items in a Python tuple need not be of same data type.

Following are some examples of Python tuples –

```
tup1 = ("Rohan", "Physics", 21, 69.75)
tup2 = (1, 2, 3, 4, 5)
```

```
tup3 = ("a", "b", "c", "d")

tup4 = (25.50, True, -55, 1+2j)
```

In Python, tuple is a sequence data type. It is an ordered collection of items. Each item in the tuple has a unique position index, starting from 0.

In C/C++/Java array, the array elements must be of same type. On the other hand, Python tuple may have objects of different data types.

Python tuple and list both are sequences. One major difference between the two is, Python list is mutable, whereas tuple is immutable. Although any item from the tuple can be accessed using its index, and cannot be modified, removed or added.

Python Tuple Operations

In Python, Tuple is a sequence. Hence, we can concatenate two tuples with + operator and concatenate multiple copies of a tuple with "*" operator. The membership operators "in" and "not in" work with tuple object.

Python Expression	Results	Description
(1, 2, 3) + (4, 5, 6)	(1, 2, 3, 4, 5, 6)	Concatenation
('Hi!',) * 4	('Hi!', 'Hi!', 'Hi!', 'Hi!')	Repetition
3 in (1, 2, 3)	True	Membership

Note that even if there is only one object in a tuple, you must give a comma after it. Otherwise, it is treated as a string.

Python - Access Tuple Items

In Python, Tuple is a sequence. Each object in the list is accessible with its index. The index starts from "0". Index or the last item in the tuple is "length-1". To access values in tuples, use the square brackets for slicing along with the index or indices to obtain value available at that index.

The slice operator fetches one or more items from the tuple.

```
obj = tup1(i)
```

Example 1

Put the index inside square brackets to retrieve the item at its position.

```
tup1 = ("Rohan", "Physics", 21, 69.75)
tup2 = (1, 2, 3, 4, 5)

print ("Item at 0th index in tup1tup2: ", tup1[0])
print ("Item at index 2 in list2: ", tup2[2])
```

It will produce the following **output** −

```
Item at 0th index in tup1: Rohan
Item at index 2 in tup2: 3
```

Example 2

Python allows negative index to be used with any sequence type. The "-1" index refers to the last item in the tuple.

```
tup1 = ("a", "b", "c", "d")
tup2 = (25.50, True, -55, 1+2j)
print ("Item at 0th index in tup1: ", tup1[-1])
print ("Item at index 2 in tup2: ", tup2[-3])
```

It will produce the following **output** −

```
Item at 0th index in tup1: d
Item at index 2 in tup2: True
```

Extracting a Subtuple from a Tuple

The slice operator extracts a subtuple from the original tuple.

Subtup = tup1[i:j]

Parameters

- **i** − index of the first item in the subtup
- **j** − index of the item next to the last in the subtup

This will return a slice from ith to (j-1)th items from the $_{\text{tup1}}$.

Example 3

Take a look at the following example −

```
tup1 = ("a", "b", "c", "d")
tup2 = (25.50, True, -55, 1+2j)

print ("Items from index 1 to 2 in tup1: ", tup1[1:3])
print ("Items from index 0 to 1 in tup2: ", tup2[0:2])
```

It will produce the following **output** −

```
Items from index 1 to 2 in tup1: ('b', 'c')
Items from index 0 to 1 in tup2: (25.5, True)
```

Example 4

While slicing, both operands "i" and "j" are optional. If not used, "i" is 0 and "j" is the last item in the tuple. Negative index can be used in slicing. See the following example −

```
tup1 = ("a", "b", "c", "d")
tup2 = (25.50, True, -55, 1+2j)
tup4 = ("Rohan", "Physics", 21, 69.75)
tup3 = (1, 2, 3, 4, 5)

print ("Items from index 1 to last in tup1: ", tup1[1:])
print ("Items from index 0 to 1 in tup2: ", tup2[:2])
print ("Items from index 2 to last in tup3", tup3[2:-1])
print ("Items from index 0 to index last in tup4", tup4[:])
```

It will produce the following **output** −

```
Items from index 1 to last in tup1: ('b', 'c', 'd')
Items from index 0 to 1 in tup2: (25.5, True)
Items from index 2 to last in tup3: (3, 4)
Items from index 0 to index last in tup4: ('Rohan', 'Physics', 21, 69.75)
```

Python - Update Tuples

In Python, tuple is an immutable data type. An immutable object cannot be modified once it is created in the memory.

Example 1

If we try to assign a new value to a tuple item with slice operator, Python raises TypeError. See the following example −

```
tup1 = ("a", "b", "c", "d")
tup1[2] = 'Z'
print ("tup1: ", tup1)
```

It will produce the following **output** −

```
Traceback (most recent call last):
 File "C:\Users\mlath\examples\main.py", line 2, in <module>
  tup1[2] = 'Z'
  ~~~~^^^
TypeError: 'tuple' object does not support item assignment
```

Hence, it is not possible to update a tuple. Therefore, the tuple class doesn't provide methods for adding, inserting, deleting, sorting items from a tuple object, as the list class.

How to Update a Python Tuple?

You can use a work-around to update a tuple. Using the list() function, convert the tuple to a list, perform the desired append/insert/remove operations and then parse the list back to tuple object.

Example 2

Here, we convert the tuple to a list, update an existing item, append a new item and sort the list. The list is converted back to tuple.

```
tup1 = ("a", "b", "c", "d")
print ("Tuple before update", tup1, "id(): ", id(tup1))

list1 = list(tup1)
list1[2]='F'
list1.append('Z')
list1.sort()
print ("updated list", list1)

tup1 = tuple(list1)
print ("Tuple after update", tup1, "id(): ", id(tup1))
```

It will produce the following **output** −

```
Tuple before update ('a', 'b', 'c', 'd') id(): 2295023084192
updated list ['F', 'Z', 'a', 'b', 'd']
Tuple after update ('F', 'Z', 'a', 'b', 'd') id(): 2295021518128
```

However, note that the id() of tup1 before update and after update are different. It means that a new tuple object is created and the original tuple object is not modified in-place.

Python - Unpack Tuple Items

The term "unpacking" refers to the process of parsing tuple items in individual variables. In Python, the parentheses are the default delimiters for a literal representation of sequence object.

Following statements to declare a tuple are identical.

```
>>> t1 = (x,y)
>>> t1 = x,y
>>> type (t1)
<class 'tuple'>
```

Example 1

To store tuple items in individual variables, use multiple variables on the left of assignment operator, as shown in the following example −

```
tup1 = (10,20,30)
x, y, z = tup1
print ("x: ", x, "y: ", "z: ",z)
```

It will produce the following **output** −

```
x: 10 y: 20 z: 30
```

That's how the tuple is unpacked in individual variables.

Using to Unpack a Tuple

In the above example, the number of variables on the left of assignment operator is equal to the items in the tuple. What if the number is not equal to the items?

Example 2

If the number of variables is more or less than the length of tuple, Python raises a ValueError.

```
tup1 = (10,20,30)
x, y = tup1
x, y, p, q = tup1
```

It will produce the following **output** −

```
  x, y = tup1
  ^^^^
ValueError: too many values to unpack (expected 2)
  x, y, p, q = tup1
  ^^^^^^^^^^
ValueError: not enough values to unpack (expected 4, got 3)
```

In such a case, the "*" symbol is used for unpacking. Prefix "*" to "y", as shown below −

```
tup1 = (10,20,30)
```

```
x, *y = tup1
print ("x: ", "y: ", y)
```

It will produce the following **output** −

```
x: y: [20, 30]
```

The first value in tuple is assigned to "x", and rest of items to "y" which becomes a list.

Example 3

In this example, the tuple contains 6 values and variables to be unpacked are 3. We prefix "*" to the second variable.

```
tup1 = (10,20,30, 40, 50, 60)
x, *y, z = tup1
print ("x: ",x, "y: ", y, "z: ", z)
```

It will produce the following **output** −

```
x: 10 y: [20, 30, 40, 50] z: 60
```

Here, values are unpacked in "x" and "z" first, and then the rest of values are assigned to "y" as a list.

Example 4

What if we add "*" to the first variable?

```
tup1 = (10,20,30, 40, 50, 60)
*x, y, z = tup1
print ("x: ",x, "y: ", y, "z: ", z)
```

It will produce the following **output** −

```
x: [10, 20, 30, 40] y: 50 z: 60
```

Here again, the tuple is unpacked in such a way that individual variables take up the value first, leaving the remaining values to the list "x".

Python - Loop Tuples

You can traverse the items in a tuple with Python's **for** loop construct. The traversal can be done, using tuple as an iterator or with the help of index.

Syntax

Python tuple gives an iterator object. To iterate a tuple, use the **for** statement as follows −

```
for obj in tuple:
   . . .
   . . .
```

Example 1

The following example shows a simple Python **for** loop construct −

```
tup1 = (25, 12, 10, -21, 10, 100)
for num in tup1:
   print (num, end = ' ')
```

It will produce the following **output** −

```
25 12 10 -21 10 100
```

Example 2

To iterate through the items in a tuple, obtain the range object of integers "0" to "len-1".

```
tup1 = (25, 12, 10, -21, 10, 100)
indices = range(len(tup1))
for i in indices:
   print ("tup1[{}]: ".format(i), tup1[i])
```

It will produce the following **output** −

```
tup1[0]: 25
tup1 [1]: 12
tup1 [2]: 10
tup1 [3]: -21
tup1 [4]: 10
tup1 [5]: 100
```

Python - Join Tuples

In Python, a Tuple is classified as a sequence type object. It is a collection of items, which may be of different data types, with each item having a positional index starting with 0. Although this definition also applies to a list, there are two major differences in list and tuple. First, while items are placed in square brackets in case of List (example: [10,20,30,40]), the tuple is formed by putting the items in parentheses (example: (10,20,30,40)).

In Python, a Tuple is an immutable object. Hence, it is not possible to modify the contents of a tuple one it is formed in the memory.

However, you can use different ways to join two Python tuples.

Example 1

All the sequence type objects support concatenation operator, with which two lists can be joined.

```
T1 = (10,20,30,40)
T2 = ('one', 'two', 'three', 'four')
T3 = T1+T2
print ("Joined Tuple:", T3)
```

It will produce the following **output** −

```
Joined Tuple: (10, 20, 30, 40, 'one', 'two', 'three', 'four')
```

Example 2

You can also use the augmented concatenation operator with the "+=" symbol to append T2 to T1

```
T1 = (10,20,30,40)
T2 = ('one', 'two', 'three', 'four')
T1+=T2
print ("Joined Tuple:", T1)
```

Example 3

The same result can be obtained by using the extend() method. Here, we need cast the two tuple objects to lists, extend so as to add elements from one list to another, and convert the joined list back to a tuple.

```
T1 = (10,20,30,40)
T2 = ('one', 'two', 'three', 'four')
```

```
L1 = list(T1)
L2 = list(T2)
L1.extend(L2)
T1 = tuple(L1)
print ("Joined Tuple:", T1)
```

Example 4

Python's built-in sum() function also helps in concatenating tuples. We use an expression

```
sum((t1, t2), ())
```

The elements of the first tuple are appended to an empty tuple first, and then elements from second tuple are appended and returns a new tuple that is concatenation of the two.

```
T1 = (10,20,30,40)
T2 = ('one', 'two', 'three', 'four')
T3 = sum((T1, T2), ())
print ("Joined Tuple:", T3)
```

Example 5

A slightly complex approach for merging two tuples is using list comprehension, as following code shows −

```
T1 = (10,20,30,40)
T2 = ('one', 'two', 'three', 'four')
L1, L2 = list(T1), list(T2)
L3 = [y for x in [L1, L2] for y in x]
T3 = tuple(L3)
print ("Joined Tuple:", T3)
```

Example 6

You can run a **for** loop on the items in second loop, convert each item in a single item tuple and concatenate it to first tuple with the "+=" operator

```
T1 = (10,20,30,40)
T2 = ('one', 'two', 'three', 'four')
for t in T2:
  T1+=(t,)
print (T1)
```

Python - Tuple Methods

Since a tuple in Python is immutable, the tuple class doesn't define methods for adding or removing items. The tuple class defines only two methods.

Sr.No	Methods & Description
1	**tuple.count(obj)** Returns count of how many times obj occurs in tuple
2	**tuple.index(obj)** Returns the lowest index in tuple that obj appears

Finding the Index of a Tuple Item

The index() method of tuple class returns the index of first occurrence of the given item.

Syntax

tuple.index(obj)

Return value

The index() method returns an integer, representing the index of the first occurrence of "obj".

Example

Take a look at the following example −

```
tup1 = (25, 12, 10, -21, 10, 100)
print ("Tup1:", tup1)
x = tup1.index(10)
print ("First index of 10:", x)
```

It will produce the following **output** −

```
Tup1: (25, 12, 10, -21, 10, 100)
First index of 10: 2
```

Counting Tuple Items

The count() method in tuple class returns the number of times a given object occurs in the tuple.

Syntax

tuple.count(obj)

Return Value

Number of occurrence of the object. The count() method returns an integer.

Example

```
tup1 = (10, 20, 45, 10, 30, 10, 55)
print ("Tup1:", tup1)
c = tup1.count(10)
print ("count of 10:", c)
```

It will produce the following **output** −

```
Tup1: (10, 20, 45, 10, 30, 10, 55)
count of 10: 3
```

Example

Even if the items in the tuple contain expressions, they will be evaluated to obtain the count.

```
Tup1 = (10, 20/80, 0.25, 10/40, 30, 10, 55)
print ("Tup1:", tup1)
c = tup1.count(0.25)
print ("count of 10:", c)
```

It will produce the following **output** −

```
Tup1: (10, 0.25, 0.25, 0.25, 30, 10, 55)
count of 10: 3
```

Assignment- Tuple

1) Python program to find unique numbers in a tuple
2) Python program to find sum of all numbers in a tuple
3) Python program to create a tuple of 5 random integers
4) Python program to find numbers common in two tuples.
5) Python program to prepare a tuple of non-numeric items from a given tuple.

Python – Dictionaries

Dictionary is one of the built-in data types in Python. Python's dictionary is example of mapping type. A mapping object 'maps' value of one object with another.

In a language dictionary we have pairs of word and corresponding meaning. Two parts of pair are key (word) and value (meaning). Similarly, Python dictionary is also a collection of key:value pairs. The pairs are separated by comma and put inside curly brackets {}.

To establish mapping between key and value, the colon ':' symbol is put between the two.

Given below are some examples of Python dictionary objects –

```
capitals = {"Maharashtra":"Mumbai", "Gujarat":"Gandhinagar",
"Telangana":"Hyderabad", "Karnataka":"Bengaluru"}
numbers = {10:"Ten", 20:"Twenty", 30:"Thirty",40:"Forty"}

marks = {"Savita":67, "Imtiaz":88, "Laxman":91, "David":49}
```

Example 1

Only a number, string or tuple can be used as key. All of them are immutable. You can use an object of any type as the value. Hence following definitions of dictionary are also valid −

```
d1 = {"Fruit":["Mango","Banana"], "Flower":["Rose", "Lotus"]}
d2 = {('India, USA'):'Countries', ('New Delhi', 'New York'):'Capitals'}
print (d1)
print (d2)
```

It will produce the following **output** −

```
{'Fruit': ['Mango', 'Banana'], 'Flower': ['Rose', 'Lotus']}
{'India, USA': 'Countries', ('New Delhi', 'New York'): 'Capitals'}
```

Example 2

Python doesn't accept mutable objects such as list as key, and raises TypeError.

```
d1 = {["Mango","Banana"]:"Fruit", "Flower":["Rose", "Lotus"]}
print (d1)
```

It will raise a TypeError –

```
Traceback (most recent call last):
  File "C:\Users\Sairam\PycharmProjects\pythonProject\main.py", line 8, in <module>
d1 = {["Mango","Banana"]:"Fruit", "Flower":["Rose", "Lotus"]}
     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
TypeError: unhashable type: 'list'
```

Example 3

You can assign a value to more than one keys in a dictionary, but a key cannot appear more than once in a dictionary.

```
d1 = {"Banana":"Fruit", "Rose":"Flower", "Lotus":"Flower", "Mango":"Fruit"}
d2 = {"Fruit":"Banana","Flower":"Rose", "Fruit":"Mango", "Flower":"Lotus"}
print (d1)

print (d2)
```

It will produce the following **output** −

```
{'Banana': 'Fruit', 'Rose': 'Flower', 'Lotus': 'Flower', 'Mango': 'Fruit'}
{'Fruit': 'Mango', 'Flower': 'Lotus'}
```

Python Dictionary Operators

In Python, following operators are defined to be used with dictionary operands. In the example, the following dictionary objects are used.

```
d1 = {'a': 2, 'b': 4, 'c': 30}
d2 = {'a1': 20, 'b1': 40, 'c1': 60}
```

Operator	Description	Example
dict[key]	Extract/assign the value mapped with key	print (d1['b']) retrieves 4 d1['b'] = 'Z' assigns new value to key 'b'

dict1|dict2	Union of two dictionary objects, retuning new object	d3=d1|d2 ; print (d3)
		{'a': 2, 'b': 4, 'c': 30, 'a1': 20, 'b1': 40, 'c1': 60}

dict1|=dict2	Augmented dictionary union operator	d1|=d2; print (d1) {'a': 2, 'b': 4, 'c': 30, 'a1': 20, 'b1': 40, 'c1': 60}

Python - Access Dictionary Items

Using the "[ ]" Operator

A dictionary in Python is not a sequence, as the elements in dictionary are not indexed. Still, you can use the square brackets "[ ]" operator to fetch the value associated with a certain key in the dictionary object.

Example 1

```
capitals = {"Maharashtra":"Mumbai", "Gujarat":"Gandhinagar", "Telangana":"Hyderabad", "Karnataka":"Bengaluru"}
print ("Capital of Gujarat is : ", capitals['Gujarat'])
print ("Capital of Karnataka is : ", capitals['Karnataka'])
```

It will produce the following **output** −

```
Capital of Gujarat is: Gandhinagar
Capital of Karnataka is: Bengaluru
```

Example 2

Python raises a KeyError if the key given inside the square brackets is not present in the dictionary object.

```
capitals = {"Maharashtra":"Mumbai", "Gujarat":"Gandhinagar", "Telangana":"Hyderabad",
"Karnataka":"Bengaluru"}
print ("Captial of Haryana is : ", capitals['Haryana'])
```

It will produce the following **output** −

```
   print ("Captial of Haryana is : ", capitals['Haryana'])
                                       ~~~~~~~~^^^^^^^^^^^^
KeyError: 'Haryana'
```

Using the get() Method

The get() method in Python's dict class returns the value mapped to the given key.

Syntax

```
Val = dict.get("key")
```

Parameters

- **key** − An immutable object used as key in the dictionary object

Return Value

The get() method returns the object mapped with the given key.

Example 3

```
capitals = {"Maharashtra":"Mumbai", "Gujarat":"Gandhinagar", "Telangana":"Hyderabad",
"Karnataka":"Bengaluru"}
print ("Capital of Gujarat is: ", capitals.get('Gujarat'))
print ("Capital of Karnataka is: ", capitals.get('Karnataka'))
```

It will produce the following **output** −

```
Capital of Gujarat is: Gandhinagar
Capital of Karnataka is: Bengaluru
```

Example 4

Unlike the "[]" operator, the get() method doesn't raise error if the key is not found; it return None.

```
capitals = {"Maharashtra":"Mumbai", "Gujarat":"Gandhinagar", "Telangana":"Hyderabad",
"Karnataka":"Bengaluru"}
print ("Capital of Haryana is : ", capitals.get('Haryana'))
```

It will produce the following **output** −

```
Capital of Haryana is : None
```

Example 5

The get() method accepts an optional string argument. If it is given, and if the key is not found, this string becomes the return value.

```
capitals = {"Maharashtra":"Mumbai", "Gujarat":"Gandhinagar", "Telangana":"Hyderabad",
"Karnataka":"Bengaluru"}
print ("Capital of Haryana is : ", capitals.get('Haryana', 'Not found'))
```

It will produce the following **output** −

```
Capital of Haryana is: Not found
```

Python - Change Dictionary Items

Apart from the literal representation of dictionary, where we put comma-separated key:value pairs in curly brackets, we can create dictionary object with built-in dict() function.

Empty Dictionary

Using dict() function without any arguments creates an empty dictionary object. It is equivalent to putting nothing between curly brackets.

Example

```
d1 = dict()
d2 = {}
print ('d1: ', d1)
print ('d2: ', d2)
```

It will produce the following **output** −

```
d1: {}
d2: {}
```

Dictionary from List of Tuples

The dict() function constructs a dictionary from a list or tuple of two-item tuples. First item in a tuple is treated as key, and the second as its value.

Example

```
d1=dict([('a', 100), ('b', 200)])
d2 = dict((('a', 'one'), ('b', 'two')))
```

```
print ('d1: ', d1)
print ('d2: ', d2)
```

It will produce the following **output** −

```
d1: {'a': 100, 'b': 200}
d2: {'a': 'one', 'b': 'two'}
```

Dictionary from Keyword Arguments

The dict() function can take any number of keyword arguments with name=value pairs. It returns a dictionary object with the name as key and associates it to the value.

Example

```
d1=dict(a= 100, b=200)
d2 = dict(a='one', b='two')
print ('d1: ', d1)
print ('d2: ', d2)
```

It will produce the following **output** −

```
d1: {'a': 100, 'b': 200}
d2: {'a': 'one', 'b': 'two'}
```

Python - Add Dictionary Items

Using the Operator

The "[]" operator (used to access value mapped to a dictionary key) is used to update an existing key-value pair as well as add a new pair.

Syntax

```
dict["key"] = val
```

If the key is already present in the dictionary object, its value will be updated to val. If the key is not present in the dictionary, a new key-value pair will be added.

Example

In this example, the marks of "Laxman" are updated to 95.

```
marks = {"Savita":67, "Imtiaz":88, "Laxman":91, "David":49}
print ("marks dictionary before update: ", marks)
```

```
marks['Laxman'] = 95
print ("marks dictionary after update: ", marks)
```

It will produce the following **output** −

```
marks dictionary before update: {'Savita': 67, 'Imtiaz': 88, 'Laxman': 91, 'David': 49}
marks dictionary after update: {'Savita': 67, 'Imtiaz': 88, 'Laxman': 95, 'David': 49 }
```

Example

However, an item with 'Krishnan' as its key is not available in the dictionary, hence a new key-value pair is added.

```
marks = {"Savita":67, "Imtiaz":88, "Laxman":91, "David":49}
print ("marks dictionary before update: ", marks)
marks['Krishan'] = 74
print ("marks dictionary after update: ", marks)
```

It will produce the following **output** −

```
marks dictionary before update: {'Savita': 67, 'Imtiaz': 88, 'Laxman': 91, 'David': 49}
marks dictionary after update: {'Savita': 67, 'Imtiaz': 88, 'Laxman': 91, 'David': 49 , 'Krishan': 74}
```

Using the update() Method

You can use the update() method in dict class in three different ways:

Update with Another Dictionary

In this case, the update() method's argument is another dictionary. Value of keys common in both dictionaries is updated. For new keys, key-value pair is added in the existing dictionary

Syntax

```
d1.update(d2)
```

Return value

The existing dictionary is updated with new key-value pairs added to it.

Example

```
marks = {"Savita":67, "Imtiaz":88, "Laxman":91, "David":49}
print ("marks dictionary before update: \n", marks)
marks1 = {"Sharad": 51, "Mushtaq": 61, "Laxman": 89}
marks.update(marks1)
print ("marks dictionary after update: \n", marks)
```

It will produce the following **output** −

```
marks dictionary before update:
{'Savita': 67, 'Imtiaz': 88, 'Laxman': 91, 'David': 49}
marks dictionary after update:
{'Savita': 67, 'Imtiaz': 88, 'Laxman': 89, 'David': 49, 'Sharad': 51, 'Mushtaq': 61}
```

Update with Iterable

If the argument to update() method is a list or tuple of two item tuples, an item each for it is added in the existing dictionary, or updated if the key is existing.

Syntax

```
d1.update([(k1, v1), (k2, v2)])
```

Return value

Existing dictionary is updated with new keys added.

Example

```
marks = {"Savita":67, "Imtiaz":88, "Laxman":91, "David":49}
print ("marks dictionary before update: \n", marks)
marks1 = [("Sharad", 51), ("Mushtaq", 61), ("Laxman", 89)]
marks.update(marks1)
print ("marks dictionary after update: \n", marks)
```

It will produce the following **output** −

```
marks dictionary before update:
 {'Savita': 67, 'Imtiaz': 88, 'Laxman': 91, 'David': 49}
marks dictionary after update:
 {'Savita': 67, 'Imtiaz': 88, 'Laxman': 89, 'David': 49, 'Sharad': 51, 'Mushtaq': 61}
```

Update with Keyword Arguments

Third version of update() method accepts list of keyword arguments in name=value format. New k-v pairs are added, or value of existing key is updated.

Syntax

```
d1.update(k1=v1, k2=v2)
```

Return value

Existing dictionary is updated with new key-value pairs added.

Example

```
marks = {"Savita":67, "Imtiaz":88, "Laxman":91, "David":49}
print ("marks dictionary before update: \n", marks)
```

```
marks.update(Sharad = 51, Mushtaq = 61, Laxman = 89)
print ("marks dictionary after update: \n", marks)
```

It will produce the following **output** −

```
marks dictionary before update:
 {'Savita': 67, 'Imtiaz': 88, 'Laxman': 91, 'David': 49}
marks dictionary after update:
 {'Savita': 67, 'Imtiaz': 88, 'Laxman': 89, 'David': 49, 'Sharad': 51, 'Mushtaq': 61}
```

Using the Unpack Operator

The "**" symbol prefixed to a dictionary object unpacks it to a list of tuples, each tuple with key and value. Two **dict** objects are unpacked and merged together and obtain a new dictionary.

Syntax

```
d3 = {**d1, **d2}
```

Return value

Two dictionaries are merged and a new object is returned.

Example

```
marks = {"Savita":67, "Imtiaz":88, "Laxman":91, "David":49}
print ("marks dictionary before update: \n", marks)
marks1 = {"Sharad": 51, "Mushtaq": 61, "Laxman": 89}
newmarks = {**marks, **marks1}
print ("marks dictionary after update: \n", newmarks)
```

It will produce the following **output** −

```
marks dictionary before update:
 {'Savita': 67, 'Imtiaz': 88, 'Laxman': 91, 'David': 49}
marks dictionary after update:
 {'Savita': 67, 'Imtiaz': 88, 'Laxman': 89, 'David': 49, 'Sharad': 51, 'Mushtaq': 61}
```

Using the Union Operator (|)

Python introduces the "|" (pipe symbol) as the union operator for dictionary operands. It updates existing keys in dict object on left, and adds new key-value pairs to return a new dict object.

Syntax

```
d3 = d1 | d2
```

Return value

The Union operator return a new dict object after merging the two dict operands

Example

```
marks = {"Savita":67, "Imtiaz":88, "Laxman":91, "David":49}
print ("marks dictionary before update: \n", marks)
marks1 = {"Sharad": 51, "Mushtaq": 61, "Laxman": 89}
newmarks = marks | marks1
print ("marks dictionary after update: \n", newmarks)
```

It will produce the following **output** −

```
marks dictionary before update:
 {'Savita': 67, 'Imtiaz': 88, 'Laxman': 91, 'David': 49}
marks dictionary after update:
 {'Savita': 67, 'Imtiaz': 88, 'Laxman': 89, 'David': 49, 'Sharad': 51, 'Mushtaq': 61}
```

Using "|=" Operator

The "|=" operator is an augmented Union operator. It performs in-place update o n the dictionary operand on left by adding new keys in the operand on right, and updating the existing keys.

Syntax

```
d1 |= d2
```

Example

```
marks = {"Savita":67, "Imtiaz":88, "Laxman":91, "David":49}
print ("marks dictionary before update: \n", marks)
marks1 = {"Sharad": 51, "Mushtaq": 61, "Laxman": 89}
marks |= marks1
print ("marks dictionary after update: \n", marks)
```

It will produce the following **output** −

```
marks dictionary before update:
 {'Savita': 67, 'Imtiaz': 88, 'Laxman': 91, 'David': 49}
marks dictionary after update:
 {'Savita': 67, 'Imtiaz': 88, 'Laxman': 89, 'David': 49, 'Sharad': 51, 'Mushtaq': 61}
```

Python - Remove Dictionary Items

Using del Keyword

Python's **del** keyword deletes any object from the memory. Here we use it to delete a key-value pair in a dictionary.

Syntax

```
del dict['key']
```

Example

```
numbers = {10:"Ten", 20:"Twenty", 30:"Thirty",40:"Forty"}
print ("numbers dictionary before delete operation: \n", numbers)
del numbers[20]
print ("numbers dictionary before delete operation: \n", numbers)
```

It will produce the following **output** −

```
numbers dictionary before delete operation:
 {10: 'Ten', 20: 'Twenty', 30: 'Thirty', 40: 'Forty'}
numbers dictionary before delete operation:
 {10: 'Ten', 30: 'Thirty', 40: 'Forty'}
```

Example

The del keyword with the dict object itself removes it from memory.

```
numbers = {10:"Ten", 20:"Twenty", 30:"Thirty",40:"Forty"}
print ("numbers dictionary before delete operation: \n", numbers)
del numbers
print ("numbers dictionary before delete operation: \n", numbers)
```

It will produce the following **output** −

```
numbers dictionary before delete operation:
 {10: 'Ten', 20: 'Twenty', 30: 'Thirty', 40: 'Forty'}
Traceback (most recent call last):
 File "C:\Users\mlath\examples\main.py", line 5, in <module>
  print ("numbers dictionary before delete operation: \n", numbers)
                                                           ^^^^^^^
NameError: name 'numbers' is not defined
```

Using pop() Method

The pop() method of dict class causes an element with the specified key to be removed from the dictionary.

Syntax

```
val = dict.pop(key)
```

Return value

The pop() method returns the value of the specified key after removing the key-value pair.

Example

```
numbers = {10:"Ten", 20:"Twenty", 30:"Thirty",40:"Forty"}
print ("numbers dictionary before pop operation: \n", numbers)
val = numbers.pop(20)
print ("nubvers dictionary after pop operation: \n", numbers)
print ("Value popped: ", val)
```

It will produce the following **output** −

```
numbers dictionary before pop operation: 
 {10: 'Ten', 20: 'Twenty', 30: 'Thirty', 40: 'Forty'}
nubvers dictionary after pop operation: 
 {10: 'Ten', 30: 'Thirty', 40: 'Forty'}
Value popped:  Twenty
```

Using popitem() Method

The popitem() method in dict() class doesn't take any argument. It pops out the last inserted key-value pair, and returns the same as a tuple

Syntax

```
val = dict.popitem()
```

Return Value

The popitem() method return a tuple contain key and value of the removed item from the dictionary

Example

```
numbers = {10:"Ten", 20:"Twenty", 30:"Thirty",40:"Forty"}
print ("numbers dictionary before pop operation: \n", numbers)
val = numbers.popitem()
print ("numbers dictionary after pop operation: \n", numbers)
print ("Value popped: ", val)
```

It will produce the following **output** −

```
numbers dictionary before pop operation: 
 {10: 'Ten', 20: 'Twenty', 30: 'Thirty', 40: 'Forty'}
numbers dictionary after pop operation: 
 {10: 'Ten', 20: 'Twenty', 30: 'Thirty'}
Value popped:  (40, 'Forty')
```

Using clear() Method

The clear() method in dict class removes all the elements from the dictionary object and returns an empty object.

Syntax

```
dict.clear()
```

Example

```
numbers = {10:"Ten", 20:"Twenty", 30:"Thirty",40:"Forty"}
print ("numbers dictionary before clear method: \n", numbers)
numbers.clear()
print ("numbers dictionary after clear method: \n", numbers)
```

It will produce the following **output** −

```
numbers dictionary before clear method:
 {10: 'Ten', 20: 'Twenty', 30: 'Thirty', 40: 'Forty'}
numbers dictionary after clear method:
 {}
```

Python - Dictionary View Objects

The items(), keys() and values() methods of dict class return view objects. These views are refreshed dynamically whenever any change occurs in the contents of their source dictionary object.

items() Method

The items() method returns a dict_items view object. It contains a list of tuples, each tuple made up of respective key, value pairs.

Syntax

```
Obj = dict.items()
```

Return value

The items() method returns dict_items object which is a dynamic view of (key,value) tuples.

Example

In the following example, we first obtain the dict_items() object with items() method and check how it is dynamically updated when the dictionary object is updated.

```
numbers = {10:"Ten", 20:"Twenty", 30:"Thirty",40:"Forty"}
obj = numbers.items()
print ('type of obj: ', type(obj))
print (obj)
print ("update numbers dictionary")
numbers.update({50:"Fifty"})
print ("View automatically updated")
print (obj)
```

It will produce the following **output** −

```
type of obj: <class 'dict_items'>
dict_items([(10, 'Ten'), (20, 'Twenty'), (30, 'Thirty'), (40, 'Forty')])
update numbers dictionary
View automatically updated
dict_items([(10, 'Ten'), (20, 'Twenty'), (30, 'Thirty'), (40, 'Forty'), (50, 'Fifty')])
```

keys() Method

The keys() method of dict class returns dict_keys object which is a list of all keys defined in the dictionary. It is a view object, as it gets automatically updated whenever any update action is done on the dictionary object

Syntax

```
Obj = dict.keys()
```

Return value

The keys() method returns dict_keys object which is a view of keys in the dictionary.

Example

```
numbers = {10:"Ten", 20:"Twenty", 30:"Thirty",40:"Forty"}
obj = numbers.keys()
print ('type of obj: ', type(obj))
print (obj)
print ("update numbers dictionary")
numbers.update({50:"Fifty"})
print ("View automatically updated")
print (obj)
```

It will produce the following **output** −

```
type of obj: <class 'dict_keys'>
dict_keys([10, 20, 30, 40])
update numbers dictionary
View automatically updated
dict_keys([10, 20, 30, 40, 50])
```

values() Method

The values() method returns a view of all the values present in the dictionary. The object is of dict_value type, which gets automatically updated.

Syntax

```
Obj = dict.values()
```

Return value

The values() method returns a dict_values view of all the values present in the dictionary.

Example

```
numbers = {10:"Ten", 20:"Twenty", 30:"Thirty",40:"Forty"}
obj = numbers.values()
print ('type of obj: ', type(obj))
print (obj)
print ("update numbers dictionary")
numbers.update({50:"Fifty"})
print ("View automatically updated")
print (obj)
```

It will produce the following **output** −

```
type of obj: <class 'dict_values'>
dict_values(['Ten', 'Twenty', 'Thirty', 'Forty'])
update numbers dictionary
View automatically updated
dict_values(['Ten', 'Twenty', 'Thirty', 'Forty', 'Fifty'])
```

Python - Loop Dictionaries

Unlike a list, tuple or a string, dictionary data type in Python is not a sequence, as the items do not have a positional index. However, traversing a dictionary is still possible with different techniques.

Example 1

Running a simple **for** loop over the dictionary object traverses the keys used in it.

```
numbers = {10:"Ten", 20:"Twenty", 30:"Thirty",40:"Forty"}
for x in numbers:
  print (x)
```

It will produce the following **output** −

```
10
20
30
40
```

Example 2

Once we are able to get the key, its associated value can be easily accessed either by using square brackets operator or with get() method.

```
numbers = {10:"Ten", 20:"Twenty", 30:"Thirty",40:"Forty"}
for x in numbers:
  print (x,":",numbers[x])
```

It will produce the following **output** −

```
10 : Ten
20 : Twenty
30 : Thirty
40 : Forty
```

The items(), keys() and values() methods of dict class return the view objects dict_items, dict_keys and dict_values respectively. These objects are iterators, and hence we can run a for loop over them.

Example 3

The dict_items object is a list of key-value tuples over which a for loop can be run as follows:

```
numbers = {10:"Ten", 20:"Twenty", 30:"Thirty",40:"Forty"}
for x in numbers.items():
  print (x)
```

It will produce the following **output** −

```
(10, 'Ten')
(20, 'Twenty')
(30, 'Thirty')
(40, 'Forty')
```

Here, "x" is the tuple element from the dict_items iterator. We can further unpack this tuple in two different variables.

Example 4

```
numbers = {10:"Ten", 20:"Twenty", 30:"Thirty",40:"Forty"}
for x,y in numbers.items():
   print (x,":", y)
```

It will produce the following **output** −

```
10 : Ten
20 : Twenty
30 : Thirty
40 : Forty
```

Example 5

Similarly, the collection of keys in dict_keys object can be iterated over.

```
numbers = {10:"Ten", 20:"Twenty", 30:"Thirty",40:"Forty"}
for x in numbers.keys():
   print (x, ":", numbers[x])
```

Respective Keys and values in dict_keys and dict_values are at same index. In the following example, we have a for loop that runs from 0 to the length of the dict, and use the looping variable as index and print key and its corresponding value.

Example 6

```
numbers = {10:"Ten", 20:"Twenty", 30:"Thirty",40:"Forty"}
l = len(numbers)
for x in range(l):
   print (list(numbers.keys())[x], ":", list(numbers.values())[x])
```

The above two code snippets produce identical **output** −

```
10 : Ten
20 : Twenty
30 : Thirty
40 : Forty
```

Python - Copy Dictionaries

Since a variable in Python is merely a label or reference to an object in the memory, a simple assignment operator will not create copy of object.

Example 1

In this example, we have a dictionary "d1" and we assign it to another variable "d2". If "d1" is updated, the changes also reflect in "d2".

```
d1 = {"a":11, "b":22, "c":33}
d2 = d1
print ("id:", id(d1), "dict: ",d1)
print ("id:", id(d2), "dict: ",d2)

d1["b"] = 100
print ("id:", id(d1), "dict: ",d1)
print ("id:", id(d2), "dict: ",d2)
```

Output

```
id: 2215278891200 dict: {'a': 11, 'b': 22, 'c': 33}
id: 2215278891200 dict: {'a': 11, 'b': 22, 'c': 33}
id: 2215278891200 dict: {'a': 11, 'b': 100, 'c': 33}
id: 2215278891200 dict: {'a': 11, 'b': 100, 'c': 33}
```

To avoid this, and make a shallow copy of a dictionary, use the copy() method instead of assignment.

Example 2

```
d1 = {"a":11, "b":22, "c":33}
d2 = d1.copy()
print ("id:", id(d1), "dict: ",d1)
print ("id:", id(d2), "dict: ",d2)
d1["b"] = 100
print ("id:", id(d1), "dict: ",d1)
print ("id:", id(d2), "dict: ",d2)
```

Output

When "d1" is updated, "d2" will not change now because "d2" is the copy of dictionary object, not merely a reference.

```
id: 1586671734976 dict: {'a': 11, 'b': 22, 'c': 33}
id: 1586673973632 dict: {'a': 11, 'b': 22, 'c': 33}
id: 1586671734976 dict: {'a': 11, 'b': 100, 'c': 33}
id: 1586673973632 dict: {'a': 11, 'b': 22, 'c': 33}
```

Python - Nested Dictionaries

A Python dictionary is said to have a nested structure if value of one or more keys is another dictionary. A nested dictionary is usually employed to store a complex data structure.

The following code snippet represents a nested dictionary:

```
marklist = {
   "Mahesh" : {"Phy" : 60, "maths" : 70},
   "Madhavi" : {"phy" : 75, "maths" : 68},
   "Mitchell" : {"phy" : 67, "maths" : 71}
}
```

Example 1

You can also constitute a for loop to traverse nested dictionary, as in the previous section.

```
marklist = {
   "Mahesh" : {"Phy" : 60, "maths" : 70},
   "Madhavi" : {"phy" : 75, "maths" : 68},
   "Mitchell" : {"phy" : 67, "maths" : 71}
}
for k,v in marklist.items():
   print (k, ":", v)
```

It will produce the following **output** −

```
Mahesh : {'Phy': 60, 'maths': 70}
Madhavi : {'phy': 75, 'maths': 68}
Mitchell : {'phy': 67, 'maths': 71}
```

Example 2

It is possible to access value from an inner dictionary with [] notation or get() method.

```
print (marklist.get("Madhavi")['maths'])
obj=marklist['Mahesh']
print (obj.get('Phy'))
print (marklist['Mitchell'].get('maths'))
```

It will produce the following **output** −

```
68
60
71
```

Python - Dictionary Methods

A dictionary in Python is an object of the built-in **dict** class, which defines the following methods –

Sr.No.	**Method and Description**
1	**dict.clear()** Removes all elements of dictionary dict.
2	**dict.copy()** Returns a shallow copy of dictionary dict.
3	**dict.fromkeys()** Create a new dictionary with keys from seq and values set to value.
4	**dict.get(key, default=None)** For key key, returns value or default if key not in dictionary.
5	**dict.has_key(key)** Returns true if a given key is available in the dictionary, otherwise it returns a false.
6	**dict.items()** Returns a list of dict's (key, value) tuple pairs.
7	**dict.keys()**

	Returns list of dictionary dict's keys.
8	**dict.pop()** Removes the element with specified key from the collection
9	**dict.popitem()** Removes the last inserted key-value pair
10	**dict.setdefault(key, default=None)** Similar to get(), but will set dict[key]=default if key is not already in dict.
11	**dict.update(dict2)** Adds dictionary dict2's key-values pairs to dict.
12	**dict.values()** Returns list of dictionary dict's values.