Reading: Demonstrate Lists and Tuples

Bookmark this page

We explored how to create and access the elements of the data structures available in Python in the previous module.

We will look at how to perform operations on these data structures.

Lists

As seen earlier, Lists are just a collection of heterogenous objects.

Let's create and manipulate a list.

List

In [1]:
```
mylist = [1,2,3,4]
```

To retrieve all the elements on the list in order using for loop:

In [2]:
```
#Create a for loop where i will take values from 0 to len(mylist), which 4, 4 exclusive.
for i in range(len(mylist)):
    print(mylist[i])
```

```
1
2
3
4
```

To add one element to the list, we use the append command

In [3]:
```
mylist.append(5)
print(mylist)
```

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

List is a mutable collection, which can have heterogenous, duplicate elements which can be mutable or immutable in nature. We can add elements to a list with the append command.

To add one element to the list, we use the append command

In [3]:
```
mylist.append(5)
print(mylist)
```

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

We may often want to add a whole collection and not add one element after the other. This can be achieved using loops and other iterative operations. But the easiest is to use the extend command.

To add all the elements of another collection

In [4]:
```
mysecondlist = [6,7,8,9,10]
mylist.extend(mysecondlist)
print(mylist)
```

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

In [5]:
```
mytuple = (11,12,13,14,15)
mylist.extend(mytuple)
print(mylist)
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
```

In [6]:
```
myset = {16,17,18,19,20}
mylist.extend(myset)
print(mylist)
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]
```

We can insert an element in a list at any desired position.

Insert an element in the array

In [7]:
```
#insert "zero" in the 0th (first) position. The first parameter is the position.
mylist.insert(0,"zero")
print(mylist)
```

```
['zero', 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]
```

If you insert an element in a postion that doesn't exist it doesn't throw an error. It is inserted in the last position.

In [8]:
```
mylist.insert(25,"zero")
print(mylist)
```

```
['zero', 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 'zero']
```

When we want to remove a particular element from the list, we can use the remove command. If we want to remove an element in a particular position, we use the pop command.

Remove a particular element from the array

In [9]:
```
mylist.remove("zero")
print(mylist)
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 'zero']
```

If you notice above, we had two elements "zero". The first occurence of the element in the list is removed from the list.

Pop an element from the list and return it

In [10]:
```
#Pop the 4th element from the list
removedelement = mylist.pop(3)
print("Removed element ",removedelement)
print(mylist)
```

```
Removed element  4
[1, 2, 3, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 'zero']
```

We use the index method to find the position of an element. If we want to search between a range of positions within the list the index method provides variations. index("a"), gives the index position of 'a' in the list; index("a",10) gives the index position of 'a' in the list, but starts the search after the 10^{th} position; index("a",10,20) gives the index position of 'a' in list, but searches for a between the index position 10 and 20.

To find the index of an element in the list:

In [25]:
```
#print the index position of 3
print(mylist.index(3))

#print the index position of 3, but start searching only from index 6 (7th position)
print(mylist.index(3,6))
```

```
2

---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-25-bf8fa7cfe1e8> in <module>
      1 print(mylist.index(3))
----> 2 print(mylist.index(3,6))

ValueError: 3 is not in list
```

If you notice above we have a 3, but we don't have a 3 after the 7th position. The second print returned a ValueError. When you look for an element that is not in the list, or not in the list in the position we are looking for, it throws a value error.

We have seen the use of len function in the previous sessions. It gives the total count of elements. But when you want to count the number of occurrences of a particular element, we can use the count method.

Counting the number of times an element occurs in a list.

In [29]:
```
mythirdlist = [1,2,2,3,3,3,4,4,4,4]

#print the number of times 1 occurs in the list
print(mythirdlist.count(1))

#print the number of times 2 occurs in the list
print(mythirdlist.count(2))

#print the number of times 3 occurs in the list
print(mythirdlist.count(3))

#print the number of times 4 occurs in the list
print(mythirdlist.count(4))
```

```
1
2
3
4
```

We can sort the elements of the array in ascending or descending order. The sort method mutates the list inplace, meaning that invoking the sort method on a list will change the list and not just return a sorted list copy. Reverse is another useful method which reverses the order of all elements in a list.

Sort the list in ascending or descending order in place. The default is ascending

In [33]:
```
mylisttobesorted = ["banana","orange","pear","guava","apple"]

mylisttobesorted.sort()
print(mylisttobesorted)

mylisttobesorted.sort(reverse=True)
print(mylisttobesorted)
```

```
['apple', 'banana', 'guava', 'orange', 'pear']
['pear', 'orange', 'guava', 'banana', 'apple']
```

Reverse the elements of the list

In [36]:
```
mylisttobereverse = [3,4,1,2,5,2,6]
mylisttobereverse.reverse()
print(mylisttobereverse)
```

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

We often need to make copies of the list, which does not affect the original list in data science. We use copy method in the list object to create a shallow copy. Any changes made to the copy, will not mutate the original.

Make a shallow copy of the list. Any changes to the copy will not affect the actual list

In [38]:
```
print("Original mylist ",mylist)
mylist_copy = mylist.copy()
print("Copy mylist ",mylist_copy)
mylist_copy.append("some element")
print("Copy mylist ",mylist_copy)
print("Original mylist ",mylist)
```

```
Original mylist  [1, 2, 3, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 'zero']
Copy mylist  [1, 2, 3, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 'zero']
Copy mylist  [1, 2, 3, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 'zero', 'some element']
Original mylist  [1, 2, 3, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 'zero']
```

Tuples

Tuple, as seen earlier, is immutable. Once we create a tuple, we cannot change it. So it provides just two methods, which work exactly the same as the list.

- count
- index

```
[1] mytuple = (1,2,3,4)
```

```
[4] print(mytuple.count(6))
    print(mytuple.count(2))
```

```
0
1
```

```
[5] mytuple.index(2)
```

```
1
```

```
[6] mytuple.index(7)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-6-78b93bfb282c> in <module>()
----> 1 mytuple.index(7)

ValueError: tuple.index(x): x not in tuple
```

SEARCH STACK OVERFLOW

Set

A set is very similar to a list except that it doesn't allow duplication and sets are unordered. The **add** method in set is similar to the **append** method in list. Where it gets added is immaterial as the set is unordered. **Update** method in the set is similar to **extend** method in list. All the elements are appended to the set ensuring there are no duplicates.

```
[7] myset = {1,2,3,4,5}
```

▾ Add an element to the cell

```
[9] myset.add(6)
    print(myset)
```

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

▾ Update method on set is similar to the extend method in list. The difference is that if there are duplicate elements, one one occurence is added to the original set.

```
[10] myset2 = {4,5,6,7,8}
     myset.update(myset2)
     myset
```

```
{1, 2, 3, 4, 5, 6, 7, 8}
```

```
mylist = {1,2,9,10}
myset.update(mylist)
myset
```

```
{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
```

The set provides many methods similar to the set operations that we learn in basic high school maths.

These include -

- union
- intersection
- difference
- isSubset

These are better understood with a working example and are shown with examples in the lab session.

Dict

There is no specific method to add a key value pair to a dict. We just use the dictionary object and assign a value to the key like this – dictobj[“key1”] = “value”

The dictionary can be iterated through in two ways – based on keys or based on items the keys are mapped on to. When you choose to iterate with keys, you get the keys collection and use iterate through the collection retrieving the value mapped to the key. When you use items, the key value pair is returned as a tuple.

```
[12] mydict = {1:"one",2:"two",3:"three"}
```

```
[19] for key in mydict.keys():
       print(key," - ",mydict[key])
```

```
1  -  one
2  -  two
3  -  three
```

```
[20] for item in mydict.items():
       print(item)
```

```
(1, 'one')
(2, 'two')
(3, 'three')
```

To create an empty dictionary with keys mapping to None values or initialized to 0, we can the fromkeys() method. It returns a dictionary with the specified keys and the specified value.

```
[16] mylist1 = [1,2,3,4,5,6]

     #Creates a dictionary mapping the keys in the collection onto None
     print(mydict.fromkeys(mylist1))

     #Creates a dictionary mapping the keys on the the value passed as second parameter
     print(mydict.fromkeys(mylist1,0))
```

```
{1: None, 2: None, 3: None, 4: None, 5: None, 6: None}
{1: 0, 2: 0, 3: 0, 4: 0, 5: 0, 6: 0}
```

Like the other collection objects, dict also has copy command and it returns an identical copy of the dictionary it is invoked on. No changes done to the copy is reflected on the original.

```
[21] mydict2 = mydict.copy()
```

```
[22] mydict2["newkey"] = "newvalue"
```

```
[23] print("mydict ",mydict)
     print("mydict2 ",mydict2)
```

```
mydict  {1: 'one', 2: 'two', 3: 'three'}
mydict2  {1: 'one', 2: 'two', 3: 'three', 'newkey': 'newvalue'}
```

The lab sessions will give you some hand-on exposure to using these data structures at ease.

Post your queries on **Questionsly** and interact with AI mentors through one-one chat option.

- Home
- About Us
- Enterprise
- Blog
- Press

Support

- Contact us
- Terms of Service
- Privacy Policy