list_tuple_set

September 16, 2019

#Data Structures

In simple terms, It is the the collection or group of data in a particular structure.

##Lists

Lists are the most commonly used data structure. Think of it as a sequence of data that is enclosed in square brackets and data are separated by a comma. Each of these data can be accessed by calling it's index value.

Lists are declared by just equating a variable to '[ ]' or list.

```
In [1]: a = []

In [2]: print(type(a))

<class 'list'>
```

One can directly assign the sequence of data to a list x as shown.

```
In [3]: x = ['apple', 'orange']
```

0.0.1 Indexing

In python, Indexing starts from 0. Thus now the list x, which has two elements will have apple at 0 index and orange at 1 index.

```
In [4]: x[0]

Out[4]: 'apple'

In [5]: x[1]

Out[5]: 'orange'
```

Indexing can also be done in reverse order. That is the last element can be accessed first. Here, indexing starts from -1. Thus index value -1 will be orange and index -2 will be apple.

```
In [6]: x[-1]

Out[6]: 'orange'
```

```
In [7]: x[-2]

Out[7]: 'apple'
```

As you might have already guessed, x[0] = x[-2], x[1] = x[-1]. This concept can be extended towards lists with more many elements.

```
In [8]: y = ['carrot','potato']
```

Here we have declared two lists x and y each containing its own data. Now, these two lists can again be put into another list say z which will have it's data as two lists. This list inside a list is called as nested lists and is how an array would be declared which we will see later.

```
In [9]: z  = [x,y]
        print(z)

[['apple', 'orange'], ['carrot', 'potato']]
```

Indexing in nested lists can be quite confusing if you do not understand how indexing works in python. So let us break it down and then arrive at a conclusion.

Let us access the data 'apple' in the above nested list. First, at index 0 there is a list ['apple','orange'] and at index 1 there is another list ['carrot','potato']. Hence z[0] should give us the first list which contains 'apple'.

```
In [10]: z1 = z[0]
         print(z1)

['apple', 'orange']
```

Now observe that z1 is not at all a nested list thus to access 'apple', z1 should be indexed at 0.

```
In [11]: z1[0]

Out[11]: 'apple'
```

Instead of doing the above, In python, you can access 'apple' by just writing the index values each time side by side.

```
In [12]: z[0][0]

Out[12]: 'apple'
```

If there was a list inside a list inside a list then you can access the innermost value by executing z[ ][ ][ ].

0.0.2 Slicing

Indexing was only limited to accessing a single element, Slicing on the other hand is accessing a sequence of data inside the list. In other words "slicing" the list.

Slicing is done by defining the index values of the first element and the last element from the parent list that is required in the sliced list. It is written as parentlist[a : b] where a,b are the index values from the parent list. If a or b is not defined then the index value is considered to be the first value for a if a is not defined and the last value for b when b is not defined.

```
In [13]: num = [0,1,2,3,4,5,6,7,8,9]

In [14]: print(num[0:4])

[0, 1, 2, 3]


In [15]: print(num[4:])

[4, 5, 6, 7, 8, 9]
```

You can also slice a parent list with a fixed length or step length.

```
In [16]: num[:9:3]

Out[16]: [0, 3, 6]
```

###Built in List Functions

To find the length of the list or the number of elements in a list, **len( )** is used.

```
In [17]: len(num)

Out[17]: 10
```

If the list consists of all integer elements then **min( )** and **max( )** gives the minimum and maximum value in the list.

```
In [18]: min(num)

Out[18]: 0

In [19]: max(num)

Out[19]: 9
```

Lists can be concatenated by adding, '+' them. The resultant list will contain all the elements of the lists that were added. The resultant list will not be a nested list.

```
In [20]: [1,2,3] + [5,4,7]

Out[20]: [1, 2, 3, 5, 4, 7]
```

There might arise a requirement where you might need to check if a particular element is there in a predefined list. Consider the below list.

```
In [21]: names = ['Earth','Air','Fire','Water']
```

To check if 'Fire' and 'Rajath' is present in the list names. A conventional approach would be to use a for loop and iterate over the list and use the if condition. But in python you can use 'a in b' concept which would return 'True' if a is present in b and 'False' if not.

```
In [22]: 'Fire' in names

Out[22]: True

In [23]: 'VIT' in names

Out[23]: False
```

In a list with elements as string, **max( )** and **min( )** is applicable. **max( )** would return a string element whose ASCII value is the highest and the lowest when **min( )** is used. Note that only the first index of each element is considered each time and if they value is the same then second index considered so on and so forth.

```
In [24]: mlist = ['bzaa','ds','nc','az','z','klm']

In [25]: print(max(mlist))
         print(min(mlist))

z
az
```

Here the first index of each element is considered and thus z has the highest ASCII value thus it is returned and minimum ASCII is a. But what if numbers are declared as strings?

```
In [26]: nlist = ['1','94','93','1000','01']

In [27]: print(max(nlist))
         print(min(nlist))

94
01
```

Even if the numbers are declared in a string the first index of each element is considered and the maximum and minimum values are returned accordingly.

But if you want to find the **max( )** string element based on the length of the string then another parameter 'key=len' is declared inside the **max( )** and **min( )** function.

```
In [28]: print(max(names))
         print(min(names))
```

```
Water
Air
```

```
In [29]: print(max(names, key=len))
         print(min(names, key=len))
```

```
Earth
Air
```

But even 'Water' has length 5. **max()** or **min()** function returns the first element when there are two or more elements with the same length.

Any other built in function can be used or lambda function (will be discussed later) in place of len.

A string can be converted into a list by using the **list()** function.

```
In [30]: names = ['Air','Fire','Water','Earth']
```

```
In [31]: print(max(names, key=len))
         print(min(names, key=len))
```

```
Water
Air
```

```
In [32]: list('hello')
```

```
Out[32]: ['h', 'e', 'l', 'l', 'o']
```

append( ) is used to add a element at the end of the list.

```
In [33]: lst = [1,1,4,8,7]
```

```
In [34]: lst.append(1)
         print(lst)
```

```
[1, 1, 4, 8, 7, 1]
```

count( ) is used to count the number of a particular element that is present in the list.

```
In [35]: lst.count(1)
```

```
Out[35]: 3
```

append( ) function can also be used to add a entire list at the end. Observe that the resultant list becomes a nested list.

```
In [36]: lst1 = [5,4,2,8]
```

```
In [37]: lst.append(lst1)
         print(lst)

[1, 1, 4, 8, 7, 1, [5, 4, 2, 8]]
```

But if nested list is not what is desired then **extend( )** function can be used.

```
In [38]: lst.extend(lst1)
         print(lst)

[1, 1, 4, 8, 7, 1, [5, 4, 2, 8], 5, 4, 2, 8]
```

index( ) is used to find the index value of a particular element. Note that if there are multiple elements of the same value then the first index value of that element is returned.

```
In [39]: lst.index(1)

Out[39]: 0
```

insert(x,y) is used to insert a element y at a specified index value x. **append( )** function made it only possible to insert at the end.

```
In [40]: lst.insert(5, 'name')
         print(lst)

[1, 1, 4, 8, 7, 'name', 1, [5, 4, 2, 8], 5, 4, 2, 8]
```

insert(x,y) inserts but does not replace element. If you want to replace the element with another element you simply assign the value to that particular index.

```
In [41]: lst[5] = 'Python'
         print(lst)

[1, 1, 4, 8, 7, 'Python', 1, [5, 4, 2, 8], 5, 4, 2, 8]
```

pop( ) function return the last element in the list. This is similar to the operation of a stack. Hence it wouldn't be wrong to tell that lists can be used as a stack.

```
In [42]: lst.pop()

Out[42]: 8
```

Index value can be specified to pop a ceratin element corresponding to that index value.

```
In [43]: lst.pop(0)

Out[43]: 1
```

pop( ) is used to remove element based on it's index value which can be assigned to a variable. One can also remove element by specifying the element itself using the **remove( )** function.

```
In [44]: lst.remove('Python')
         print(lst)
```

```
[1, 4, 8, 7, 1, [5, 4, 2, 8], 5, 4, 2]
```

Alternative to **remove** function but with using index value is **del**

```
In [45]: del lst[1]
         print(lst)
```

```
[1, 8, 7, 1, [5, 4, 2, 8], 5, 4, 2]
```

The entire elements present in the list can be reversed by using the **reverse()** function.

```
In [46]: lst.reverse()
         print(lst)
```

```
[2, 4, 5, [5, 4, 2, 8], 1, 7, 8, 1]
```

Note that the nested list [5,4,2,8] is treated as a single element of the parent list lst. Thus the elements inside the nested list is not reversed.

Python offers built in operation **sort( )** to arrange the elements in ascending order.

```
In [47]: del lst[3]
         lst.sort()
         print(lst)
```

```
[1, 1, 2, 4, 5, 7, 8]
```

For descending order, By default the reverse condition will be False for reverse. Hence changing it to True would arrange the elements in descending order.

```
In [48]: lst.sort(reverse=True)
         print(lst)
```

```
[8, 7, 5, 4, 2, 1, 1]
```

Similarly for lists containing string elements, **sort( )** would sort the elements based on it's ASCII value in ascending and by specifying reverse=True in descending.

```
In [49]: names.sort()
         print(names)
         names.sort(reverse=True)
         print(names)
```

```
['Air', 'Earth', 'Fire', 'Water']
['Water', 'Fire', 'Earth', 'Air']
```

To sort based on length key=len should be specified as shown.

```
In [50]: names.sort(key=len)
         print(names)
         names.sort(key=len,reverse=True)
         print(names)
```

```
['Air', 'Fire', 'Water', 'Earth']
['Water', 'Earth', 'Fire', 'Air']
```

0.0.3 Copying a list

Most of the new python programmers commit this mistake. Consider the following,

```
In [51]: lista= [2,1,4,3]
```

```
In [52]: listb = lista
         print(listb)
```

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

Here, We have declared a list, lista = [2,1,4,3]. This list is copied to listb by assigning it's value and it get's copied as seen. Now we perform some random operations on lista.

```
In [53]: lista.pop()
         print(lista)
         lista.append(9)
         print(lista)
```

```
[2, 1, 4]
[2, 1, 4, 9]
```

```
In [54]: print(listb)
```

```
[2, 1, 4, 9]
```

listb has also changed though no operation has been performed on it. This is because you have assigned the same memory space of lista to listb. So how do fix this?

If you recall, in slicing we had seen that parentlist[a:b] returns a list from parent list with start index a and end index b and if a and b is not mentioned then by default it considers the first and last element. We use the same concept here. By doing so, we are assigning the data of lista to listb as a variable.

```
In [55]: lista = [2,1,4,3]

In [56]: listb = lista[:]
         print(listb)

[2, 1, 4, 3]


In [57]: lista.pop()
         print(lista)
         lista.append(9)
         print(lista)

[2, 1, 4]
[2, 1, 4, 9]


In [58]: print(listb)

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

Dictionaries

Dictionaries are more used like a database because here you can index a particular sequence with your user defined string.

To define a dictionary, equate a variable to { } or dict()

```
In [59]: d0 = {}
         d1 = dict()
         print(type(d0), type(d1))

<class 'dict'> <class 'dict'>
```

Dictionary works somewhat like a list but with an additional capability of assigning it's own index style.

```
In [60]: d0['One'] = 1
         d0['OneTwo'] = 12
         print(d0)

{'One': 1, 'OneTwo': 12}
```

That is how a dictionary looks like. Now you are able to access '1' by the index value set at 'One'

```
In [61]: print(d0['One'])
         print(d0['OneTwo'])
```

```
1
12
```

Two lists which are related can be merged to form a dictionary.

```
In [62]: names = ['One', 'Two', 'Three', 'Four', 'Five']
         numbers = [1, 2, 3, 4, 5]
```

zip( ) function is used to combine two lists

```
In [63]: d2 = zip(names,numbers)
         #print(list(d2))
         print(d2)

<zip object at 0x7fd34cd86cc8>
```

```
In [64]: a1 = dict(d2)
         print(a1)

{'One': 1, 'Two': 2, 'Three': 3, 'Four': 4, 'Five': 5}
```

Built-in Functions
clear( ) function is used to erase the entire database that was created.

```
In [65]: a1.clear()
         print(a1)

{}
```

Dictionary can also be built using loops.

```
In [66]: for i in range(len(names)):
             a1[names[i]] = numbers[i]

In [67]: print(a1)

{'One': 1, 'Two': 2, 'Three': 3, 'Four': 4, 'Five': 5}
```

values( ) function returns a list with all the assigned values in the dictionary.

```
In [68]: list(a1.values())

Out[68]: [1, 2, 3, 4, 5]
```

keys( ) function returns all the index or the keys to which contains the values that it was assigned to.

```
In [69]: list(a1.keys())

Out[69]: ['One', 'Two', 'Three', 'Four', 'Five']
```

items( ) is returns a list containing both the list but each element in the dictionary is inside a tuple. This is same as the result that was obtained when zip function was used.

```
In [70]: list(a1.items())

Out[70]: [('One', 1), ('Two', 2), ('Three', 3), ('Four', 4), ('Five', 5)]
```

pop( ) function is used to remove a particular element and that removed element can be assigned to a new variable. But remember only the value is stored and not the key. Because the is just a index value.

```
In [71]: a2 = a1.pop('Four')
         print(a1)

{'One': 1, 'Two': 2, 'Three': 3, 'Five': 5}


In [72]: print(a2)

4


In [73]: a3 = a1.copy()

         print('Orignal:', a1)
         print('Copied: ', a3)

Orignal: {'One': 1, 'Two': 2, 'Three': 3, 'Five': 5}
Copied:  {'One': 1, 'Two': 2, 'Three': 3, 'Five': 5}


In [74]: keys = {'a', 'e', 'i', 'o', 'u' }
         value = 'vowel'

         vowels = dict.fromkeys(keys, value)
         print(vowels)

{'o': 'vowel', 'i': 'vowel', 'u': 'vowel', 'a': 'vowel', 'e': 'vowel'}
```

##Tuples

Tuples are similar to lists but only big difference is the elements inside a list can be changed but in tuple it cannot be changed. Think of tuples as something which has to be True for a particular something and cannot be True for no other values. For better understanding, Recall **divmod()** function.

```
In [75]: xyz = divmod(10,3)
         print(xyz)
         print(type(xyz))

(3, 1)
<class 'tuple'>
```

Here the quotient has to be 3 and the remainder has to be 1. These values cannot be changed whatsoever when 10 is divided by 3. Hence divmod returns these values in a tuple.

To define a tuple, A variable is assigned to paranthesis ( ) or tuple( ).

```
In [76]: tup = ()
         tup2 = tuple()
```

If you want to directly declare a tuple it can be done by using a comma at the end of the data.

```
In [77]: 27,

Out[77]: (27,)
```

27 when multiplied by 2 yields 54, But when multiplied with a tuple the data is repeated twice.

```
In [78]: 2*(27,)

Out[78]: (27, 27)
```

Values can be assigned while declaring a tuple. It takes a list as input and converts it into a tuple or it takes a string and converts it into a tuple.

```
In [79]: tup3 = tuple([1,2,3])
         print(tup3)
         tup4 = tuple('Hello')
         print(tup4)

(1, 2, 3)
('H', 'e', 'l', 'l', 'o')
```

It follows the same indexing and slicing as Lists.

```
In [80]: print(tup3[1])
         tup5 = tup4[:3]
         print(tup5)

2
('H', 'e', 'l')
```

0.0.4 Mapping one tuple to another

```
In [81]: (a,b,c)= ('alpha','beta','gamma')

In [82]: print(a,b,c)
```

```
alpha beta gamma
```

```
In [83]: d = tuple('vitchennai')
         print(d)
```

```
('v', 'i', 't', 'c', 'h', 'e', 'n', 'n', 'a', 'i')
```

0.0.5 Built In Tuple functions

count() function counts the number of specified element that is present in the tuple.

```
In [84]: d.count('i')
```

```
Out[84]: 2
```

index() function returns the index of the specified element. If the elements are more than one then the index of the first element of that specified element is returned

```
In [85]: d.index('i')
```

```
Out[85]: 1
```

##Sets

Sets are mainly used to eliminate repeated numbers in a sequence/list. It is also used to perform some standard set operations.

Sets are declared as set() which will initialize a empty set. Also set([sequence]) can be executed to declare a set with elements

```
In [86]: set1 = set()
         print(type(set1))
```

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

```
In [87]: set0 = set([1,2,2,3,3,4])
         print(set0)
```

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

elements 2,3 which are repeated twice are seen only once. Thus in a set each element is distinct.

###Built-in Functions

```
In [88]: set1 = set([1,2,3])

In [89]: set2 = set([2,3,4,5])
```

union( ) function returns a set which contains all the elements of both the sets without repition.

```
In [90]: set1.union(set2)

Out[90]: {1, 2, 3, 4, 5}
```

add( ) will add a particular element into the set. Note that the index of the newly added element is arbitrary and can be placed anywhere not neccessarily in the end.

```
In [91]: set1.add(0)
         set1

Out[91]: {0, 1, 2, 3}
```

intersection( ) function outputs a set which contains all the elements that are in both sets.

```
In [92]: set1.intersection(set2)

Out[92]: {2, 3}
```

difference( ) function ouptuts a set which contains elements that are in set1 and not in set2.

```
In [93]: set1.difference(set2)

Out[93]: {0, 1}
```

symmetric_difference( ) function ouputs a function which contains elements that are in one of the sets.

```
In [94]: set2.symmetric_difference(set1)

Out[94]: {0, 1, 4, 5}
```

issubset( ), isdisjoint( ), issuperset( ) is used to check if the set1/set2 is a subset, disjoint or superset of set2/set1 respectively.

```
In [95]: set1.issubset(set2)

Out[95]: False

In [96]: set2.isdisjoint(set1)

Out[96]: False

In [97]: set2.issuperset(set1)

Out[97]: False
```

pop( ) is used to remove an arbitrary element in the set

```
In [98]: set1.pop()
         print(set1)
```

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

remove( ) function deletes the specified element from the set.

```
In [99]: set1.remove(2)
         set1
```

```
Out[99]: {1, 3}
```

clear( ) is used to clear all the elements and make that set an empty set.

```
In [100]: set1.clear()
          set1
```

```
Out[100]: set()
```