Python 101

Data Structures in Python (Part-2) - Lists, Tuples, Sets, Frozensets, Dictionaries

Contents

Containers

Containers are any objects in python that hold other objects.

eg. Lists, Tuples, Sets, FrozenSets, Dictionaries.

Lists, Methods and Properties

Lists as we know are iterable, mutable collection of objects.

Lists CAN be Heterogeneous but they SHOULD be used for Homogeneous elements only.

A list is represented using **[ ]** literal.

It can also be created using **list()** ← we will understand this method better when we will study about OOPs.

Lists, Methods and Properties contd..

There are primarily 4 things we must know how to do in a list. Because lists are mutable so we can do the following-

1. adding an element. (.append(), .insert(), .extend())
2. deleting an element. *(.remove(), .pop(), .clear(), .del())
3. sorting a list (.sort(), .reverse())
4. Iterable operations (len, .count(), slicing, indexing, copying)

Append an element in list

```
>>>test_list = [ ]
>>>type(test_list)
<class ‘list’>

list.append(object_to_append_in_list)
>>>test_list.append(1)
>>>test_list
[1]
>>>test_list.append(2)
>>>test_list
[1, 2] ← Append adds an item in the list in the end of the list.
```

Insert an element in list

```
>>>test_list = [1,2,3]

list.insert(index_to_insert_at, object_to_insert_in_list)

>>>test_list.insert(0,8)
>>>test_list
[8, 2, 3]

# Mind that if we gave an index can be negative or positive.
# But negative indexing doesn't work as expected
>>>test_list.insert(-1, 9)
>>>test_list
[8, 9, 3]

# In order to append something in list using insert do
>>>test_list.insert(index_that_is_out_of_bound, object_to_append)
>>>test_list.insert(99999,  4)
[8, 9, 3, 4]
```

Extend a list by an iterable

We can extend a list by an iterable eg. another list

```
>>> base_list = [1, 2, 3]
>>> list_to_extend_with = [4, 5, 6]
>>>base_list.extend(list_to_extend_with)
[1, 2, 3, 4, 5, 6]
```

Multidimensional Lists

Lists can contain any object not just integers. So lists can contain lists as well.

ie.

```
>>>x=[ [1, 2, 3, 4],
       [5, 6, 7, 8],
       [9, 10, 11, 12] ]
>>> x[0][0]
1
```

.remove in list

It removes the first occurence of the value.

```
>>> x=[1,2,3,4,1]
>>> x.remove(1)
>>>x
[2,3,4,1]
```

.clear in list

It removes all of the items from list.

```
>>> x=[1,2,3,4,1]
>>> x.clear()
>>>x
[]
```

.pop in list

It removes the last element from list.

```
>>> x=[1,2,3,4]
>>> x.pop()
4
```

Mind that pop() also returns the item that has been popped.

del in list

It removes the last element from list.

```
>>> x=[1,2,3,4]
>>> del(x[0])
```

Mind that del() is a builtin not specific to the list.
del deletes any object from the memory. No matter where it is.

Iterable operations

len - gives number of items in an iterable

```
>>> x=[1, 2, 3, 1]
>>> len(x)
4
```

count - number of occurence of iem in an iterable

```
>>> x.count(1)
2
```

Indexing -

```
>>> x[2]
3
```

slicing-

```
>>> x[1:3:1]
[2,3,1]
```

sorted([1,3,4]**)** ← sorting of a list
reverse([-3, 2, 1]) ← sorting of a list in reverse

Iterable operations

Membership check-

element **in** iterable
True / False

>>> 1 in [1,2,3,4]
True

>>> ‘x’ in [1, 2, 3, 4]
False

Tuples, Methods and Properties

Tuples are iterable and immutable collection of objects.

Tuples can Homogeneous and Heterogeneous both.

A tupleis represented using **( )** literal.

It can also be created using **tuple()** ← we will understand this method better when we will study about OOPs.

Tuples are immutable so we can not add or remove anything. They have only 2 basic operations **concatenation and iterable operations (same as lists and strings).**

Tuple concatenation

concatenation-

```
>>> tuple1 = (1,2)
>>> tuple2 = (3,4)
>>> tuple1+ tuple2
(1, 2, 3, 4) ← Mind that its a new object tuple1 and tuple2 remain unchanged.
```

Sets, methods and properties

Sets are a mutable collection of **unique** objects.

Sets are declared using { } with at least one element in them. There is nothing like a **null set literal** in python.

We can also declare a set using **set().**

It supports the operations that usual sets support in mathematics. eg. **difference, union, intersection.** We can also check if a set is other set's subset or superset.

Being mutable sets also support add, remove, pop and extension of elements.

Mutable Operations

Add elements-

```
>>> set1={1}
>>>set1.add(2)
>>>set1
{1, 2}
```

Remove elements-

```
>>> set1={1,2}
>>> set1.remove(2)
>>> set1
{1}
```

Pop elements-

```
>>> set1 = {1,2}
>>> set1.pop()
1
# Mind that set pops from front unlike lists
```

Update-

```
>>>set1.update([1,2,3,4,5]) ← the argument can be any iterable
>>>set1
{1,2,3,4,5}
```

Set operations

difference-

```
>>> set1 = {1,2,3,4}
>>> set2 = {1,3,4,5}
>>> set1 - set2
{2}
>>> set2 - set1
{5}
```

Union-

```
>>>set1.union(set2)
{1,2,3,4,5}
```

Intersection-

```
>>>set1.intersection(set2)
{1,3,4}
```

Issubset-

```
>>> set1 = {1,2,3,4}
>>> set2 = {1,2,3}
>>> set2.issubset(set1)
True ← Mind that it checks for subset not for PROPER subset.
```

Frozensets, methods and properties

Frozensets are immutable sets.

These are created by using **frozenset(iterable)**

eg. `>>> frozenset([1, 2, 3, 4, 5])`

Frozen sets can perform all the Set Operations but being immutable they can not perform mutable operations eg. add, remove or extend.

Dictionaries, methods and properties

Till now all of the data structures we saw had capability of storing elements but none can be used to establish a relation between two objects.

Dictionary is a Mutable, Mapping object type which maps hashable objects to their values.

They can be created using **{ }** literal or **dict()**.

```
{ 'key1': 'value1', 'key2': 'value2'}
```

Accessing values in dictionary

```
>>> dictionary = {1:  2, 3: 4 , 5: 6}
>>> dictionary[1]
2
>>> dictionary[3]
4
>>> dictionary[5]
6

## Getting a list of all of the keys from a dictionary
>>> dictionary.keys()
dict_keys([1, 3, 5])

## Getting a list of all of the values from a dictionary
>> dictionary.values()
dict_values([2, 4, 6])

## Getting a list of a tuples containing (key, value)
>> dictionary.items()
dict_items([(1, 2), (3, 4), (5, 6)])
```

Mutable operations on Dictionary

Add an item

```
>>> d = {}
>>> d[‘x’] = ‘y’
>>> d
{‘x’: ‘y’}
```

Deleting an item

```
>>> d= {1:2, 3:4, 5:6}
>>> del(d[1])
```

Popping a value

```
>>> d= {1:2, 3:4, 5:6}
>>> d.pop(1)
2
```

Popping an item

```
>>> d= {1:2, 3:4, 5:6}
>>> d.popitem()
(5, 6)
```

Type Conversion Between Data Structures

To list -

- **list(**any_iterable_except_dictionary**)** -> it produces list of all the values in iterable.
- **list(**dictionary**)** -> It produces list of the keys only.

To set-

- **set(**any_iterable_except_dictionary**)** -> it produces set of all the values in iterable.
- **set(**dictionary**)** -> it produces set of the keys only.

Type Conversion Between Data Structures

To frozenset-

- **frozenset(**any_iterable_except_dictionary**)** -> it produces list of all the values in iterable.
- **frozenset(**dictionary**)** -> It produces list of the keys only.

To tuple-

- **tuple(**any_iterable_except_dictionary**)** -> it produces set of all the values in iterable.
- **tuple(**dictionary**)** -> it produces set of the keys only.

Type Conversion Between Data Structures

To dictionary- There are special rules when converting something to dictionary.

Dictionaries using mapping - dict(key1=value1, key2=value2, key3=value3)

Dictionaries using tuples mapping - dict(((key1, value1), (key2, value2)))

Dictionaries using lists mapping - dict([[key1, value1], [key2, value2]])

Comprehensions

Comprehension in English means understanding something. In terms of data structures in Python Comprehensions mean ***“making process of creation of Data structures easier to understand ”.***

There are primarily two types of comprehensions-

1. List Comprehension
2. Dictionary Comprehension

But the same logic of List Comprehension can be extended to Tuple, Set and even Generators.

Comprehensions contd..

List Comprehension-

```
[ i for i in some_iterable ]
```

Nested Comprehension-

```
[ j for i in some_iterable for j in i ]
```

Comprehensions contd..

Understanding Nested List Comprehension-

Write a nested for loop as -

```
for i in some_list:

    for j in i:

        j ← do something here with j
```

We finally require the value of j to use thus we write j in starting and then we write the rest of it as it is taking it all in one line- `[ j for i in some_list for j in i ]`

Comprehensions contd..

Dictionary Comprehension-

```
{ i:j for i,j in some_iterable_mapping }
```

Set comprehension -

```
{ i for i in some_iterable }
```

Generator comprehension -

```
( i for i in some_iterable ) ← it will create a generator object for some_iterable sequence not a tuple.
```

Interview Questions

What are containers?

Containers are objects in Python that can hold other objects.

Reverse a list using silicing?

list[::-1]

What do you understand by Comprehensions?

Comprehensions are a way to make it easy to understand the creation of containers in python.

Create a tuple with a single element in Python?

(1,)

We must put the comma after a single element else it will simply evaluate it as 1.

How would you create an empty set?

set()

{} creates an empty dictionary not empty set.

What do you understand by membership operator?

in is the membership operator it checks whether a given element exists in an iterable or not.

For dictionaries it only checks if the element exists in the keys of dictionary.

What is the key mapping technique of creating a dictionary?

dict(k1=v1, k2=v2, k3=v3**)**

results in -

{k1: v1, k2:v2, k3:v3}

Create a generator corresponding to a given list without using yield?

(i for i in given_list)