3/11/2019

Cheat sheet for Python Data Structures

Gopinath Jayakumar - Architect
TESTLEAF SOFTWARE SOLUTION PRIVATE LMT

Lists

Mutable, ordered series, traditionally of the same type of object.

Advantages: Mutable and ordered. Easy to understand. Relatively efficient memory usage. Disadvantages: Searching is O(n).

To create a list, use square brackets:

```
mylist = [ ]
mylist = [1,2,3]
mylist = ['a', 'b', 'c', [1,2,3] ]   # 4 elements
```

Retrieving one element, given an index

```
x = mylist[3]
```

Membership

```
3 in mylist                    # True or False
```

From another type: Given an iterable, the “list” function returns a list:

```
list('abc')         # ['a', 'b', 'c']
list((1,2,3))       # [1,2,3]
list({1,2,3})       # [1,2,3]
```

Replacing an existing element

```
mylist = ['a', 'b', 'c']
mylist[1] = 'z'
mylist              # ['a', 'z', 'c']
```

Replacing multiple existing elements

```
mylist = ['a', 'b', 'c', 'd', 'e', 'f']
mylist[1:3] = 'xyz'            # replace indexes 1 and 2 with x, y, z
mylist                         #  ['a', 'x', 'y', 'z', 'd', 'e', 'f']
```

Adding an element to the end

```
mylist = ['a', 'b', 'c']
mylist.append('d')
mylist                    # ['a', 'b', 'c', 'd']
mylist.append([1,2,3])
mylist                    # ['a', 'b', 'c', 'd', [1,2,3]]
```

Adding multiple elements to the end

```
mylist = ['a', 'b', 'c']
mylist.extend([1,2,3])
mylist                    # ['a', 'b', 'c', 'd', 1, 2, 3]
```

Removing an element from the end

```
mylist = ['a', 'b', 'c']
mylist.pop()              # returns 'c'
mylist                    # ['a', 'b']
```

Removing an element from any index

```
mylist = ['a', 'b', 'c']
mylist.pop(0)             # returns 'a'
mylist                    # ['b', 'c']
```

Removing an element based on its value (rather than its position)

```
mylist = ['a', 'b', 'c', 'a', 'a', 'b']
mylist.remove('a')        # Remove the first 'a'
mylist                    # ['b', 'c', 'a', 'a', 'b']
```

Sorting

```
mylist = ['d', 'a', 'c', 'b']
mylist.sort()             # Returns None
mylist                    # ['a', 'b', 'c', 'd']
```

Reversing

```
mylist = ['a', 'b', 'c']
mylist.reverse()            # returns None
mylist                      # ['c', 'b', 'a']
```

Joining

```
mylist = ['a', 'b', 'c']
'*'.join(mylist)            # 'a*b*c'
'...'.join(mylist)          # 'a...b...c'
```

Iterating over the elements

```
mylist = ['a', 'b', 'c']
for item in mylist:
        print(item)
```

Iterating over the sorted elements

```
mylist = ['d', 'a', 'c', 'b']
for item in sorted(mylist):
        print(item)
```

Tuples

Immutable, ordered series traditionally containing different objects.

Advantages: Immutable and ordered. Relatively efficient memory usage (more than lists). Disadvantages: Searching is O(n). Hard to understand for many Python newcomers.

Creating

```
t = ('a', 1, [1,2,3])       # () and comma indicate tuple
t = ('a',)                  # single-element tuple requires ,!
```

From another type

```
tuple([1,2,3])              # (1,2,3)
```

Iterating over the elements

```
t = ('a', 'b', 'c')
for item in t:
        print(item)
```

Iterating over the sorted elements

```
t = ('d', 'a', 'c', 'b')
for item in sorted(t):
        print(item)
```

Dictionaries

Mutable, unordered pairs (keys and values) of objects. Keys must be hashable.

Advantages: O(1) searching for keys. Makes it easy to create trees and other hierarchical data structures. Can be used to create self-documenting code. Many problems can be described in terms of key-value pairs. Disadvantages: Only lookup by key. Uses more memory than lists and tuples. Keys must be hashable.

Creating

```
{'a':1, 'b':2, 'c':3}                     # {'a': 1, 'b': 2, 'c': 3}
```

Creating from other data

```
dict( ['a',1] , ['b',2] , ['c',3] )       # {'a': 1, 'b': 2, 'c': 3}
dict( ('a',1) , ('b',2) , ('c',3) )       # {'a': 1, 'b': 2, 'c': 3}
```

Retrieving from a key

```
d = {'a':1, 'b':2, 'c':3}
d['a']                                    # 1
d['z']                                    # raises Key Error
```

Add a key-value pair

```
d = {'a':1, 'b':2, 'c':3}
d['d'] = 100
d                         # {'a': 100, 'b': 2, 'c': 3, 'd': 100}
```

Replacing an existing value

```
d = {'a':1, 'b':2, 'c':3}
d['a'] = 100
d                         # {'a': 100, 'b': 2, 'c': 3}
```

Replacing multiple existing values

```
d = {'a':1, 'b':2 }
x = {'a':555, 'z':987}
d.update(x, y=10)         # Returns None
d                         # {'a': 555, 'b': 2, 'y': 10, 'z': 987}
```

Removing an element

```
d = {'a':1, 'b':2, 'c':3}
del(d['a'])
d                         # {'c': 3, 'b': 2}
```

Getting the keys

```
d = {'a':1, 'b':2, 'c':3}
d.keys()                  # ['a', 'c', 'b']  (Python 2)
d.keys()                  # dict_keys(['a', 'b', 'c'])  (Python 3)
```

Getting the values

```
d = {'a':1, 'b':2, 'c':3}
d.values()                # [1, 2, 3]  (Python 2)
d.values()                # dict_values([1, 2, 3])  (Python 3)
```

Iterating over the keys

```
d = {'a':1, 'b':2, 'c':3}
for k in d:
    print(f"{k} : {d[k]}")
```

Iterating over the pairs

```
d = {'a':1, 'b':2, 'c':3}
for k, v in d.items()
    print(f"{k} : {v}")
```

Iterating over the sorted keys

```
d = {'a':1, 'b':2, 'A':3}
for k in sorted(d):
    print(f"{k} : {d[k]}")
```

Sets

Mutable, unordered, unique objects. Elements must be hashable.

Advantages: Searching is O(1). Lots of useful methods.

Disadvantages: Not ordered. Elements must be hashable.

Creating

```
s = {1,2,3}                      # Python 2.7,  3.x
```

Creating from another type

```
s = set( [1,2,3] )               # From list
s = set( (1,2,3) )               # From tuple
s = set('abc')                   # From string
```

Adding a value

```
s = {1,2,3}
s.add(4)
s                          # {1,2,3,4}
s.add(4)
s                          # {1,2,3,4} — duplicates are ignored
```

Adding multiple values

```
s = {1,2,3}
s.update([3,4,5])        # Any iterable will do
s                        # {1,2,3,4,5} — duplicates ignored
```

Removing an element

```
s = {1,2,3}
s.remove(1)
s                        # {2,3}
```