

Data Structures and Sequences

```
In [2]: import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import seaborn as sns
import statsmodels as sm
```

Mutable and immutable objects

Most objects in Python, such as lists, dicts, NumPy arrays, and most user-defined types (classes), are mutable. This means that the object or values that they contain can be modified:

```
In [1]: a_list = ['foo', 2, [4, 5]]
a_list[2] = (3, 4)
a_list
```

```
Out[1]: ['foo', 2, (3, 4)]
```

Others, like strings and tuples, are immutable:

```
In [2]: In [46]: a_tuple = (3, 5, (4, 5))
In [47]: a_tuple[1] = 'four'
```

```
-----
TypeError                                Traceback (most recent call last)
<ipython-input-2-2c9bddc8679c> in <module>()
      1 a_tuple = (3, 5, (4, 5))
----> 2 a_tuple[1] = 'four'
```

```
TypeError: 'tuple' object does not support item assignment
```

Tuples

A **tuple** is a fixed-length, immutable sequence of Python objects. The easiest way to create one is with a comma-separated sequence of values:

```
In [4]: tup = 4, 5, 6
```

```
In [5]: tup
```

```
Out[5]: (4, 5, 6)
```

```
In [6]: nested_tup = (4, 5, 6), (7, 8)
```

```
In [7]: nested_tup
```

```
Out[7]: ((4, 5, 6), (7, 8))
```

You can convert any sequence or iterator to a tuple by invoking **tuple**:

```
In [9]: tuple([4, 0, 2])
```

```
Out[9]: (4, 0, 2)
```

Elements can be accessed with square brackets `[]` as with most other sequence types. As in C, C++, Java, and many other languages, sequences are **0-indexed** in Python:

```
In [18]: tup = tuple('string')
         tup[0]
```

```
Out[18]: 's'
```

```
In [19]: tup = tuple(['foo', [1, 2], True])
```

While the objects stored in a tuple may be mutable themselves, once the tuple is created it's not possible to modify which object is stored in each slot:

```
In [20]: tup[2] = False
```

```
-----
TypeError                                Traceback (most recent call last)
<ipython-input-20-b89d0c4ae599> in <module>()
----> 1 tup[2] = False
```

```
TypeError: 'tuple' object does not support item assignment
```

```
In [22]: tup[1].append(3)
```

```
In [23]: tup
```

```
Out[23]: ('foo', [1, 2, 3, 3], True)
```

```
In [36]: (4, None, 'foo') + (6, 0) + ('bar',)
```

```
Out[36]: (4, None, 'foo', 6, 0, 'bar')
```

Multiplying a tuple by an integer, as with lists, has the effect of concatenating together that many

copies of the tuple:

```
In [28]: ('foo', 'bar') * 4
```

```
Out[28]: ('foo', 'bar', 'foo', 'bar', 'foo', 'bar', 'foo', 'bar')
```

Unpacking tuples

```
In [29]: tup = (4, 5, 6)
         a, b, c = tup
```

```
In [30]: a, b, c
```

```
Out[30]: (4, 5, 6)
```

```
In [31]: tup = 4, 5, (6, 7)
         a, b, (c, d) = tup
```

```
In [33]: a, b, c, d
```

```
Out[33]: (4, 5, 6, 7)
```

A common use of variable unpacking is iterating over sequences of tuples or lists:

```
In [34]: seq = [(1, 2, 3), (4, 5, 6), (7, 8, 9)]

         for a, b, c in seq:
             print('a={0}, b={1}, c={2}'.format(a, b, c))
```

```
a=1, b=2, c=3
```

```
a=4, b=5, c=6
```

```
a=7, b=8, c=9
```

Advanced tuple unpacking to help with situations where you may want to “pluck” a few elements from the beginning of a tuple

Use the special syntax `** rest`, which is also used in function signatures to capture an arbitrarily long list of positional arguments:

```
In [42]: values = 1, 2, 3, 4, 5
```

```
In [43]: a, b, *rest = values
```

```
In [44]: a, b
```

```
Out[44]: (1, 2)
```

```
In [47]: rest
```

```
Out[47]: [3, 4, 5]
```

Python programmers will use the underscore `** _ **` for unwanted variables

```
a, b, *_ = values
```

Tuple methods

count: counts the number of occurrences of a value:

```
In [49]: a = (1, 2, 2, 2, 3, 4, 2)
```

```
In [4]: a.count(2)
```






```
-----
NameError                                Traceback (most recent call last)
<ipython-input-4-86fca892b707> in <module>()
----> 1 a.count(2)

NameError: name 'a' is not defined
```

```
In [6]: tuple = ( 'abcd', 786 , 2.23, 'john', 70.2 )
tinytuple = (123, 'john')

print (tuple)           # Prints complete List
print (tuple[0])        # Prints first element of the List
print (tuple[1:3])      # Prints elements starting from 2nd till 3rd
print (tuple[2:])        # Prints elements starting from 3rd element
print (tinytuple * 2)    # Prints List two times
print (tuple + tinytuple) # Prints concatenated Lists
```

```
('abcd', 786, 2.23, 'john', 70.2)
abcd
(786, 2.23)
(2.23, 'john', 70.2)
(123, 'john', 123, 'john')
('abcd', 786, 2.23, 'john', 70.2, 123, 'john')
```

Sr.No.	Function with Description
1	cmp(tuple1, tuple2)  Compares elements of both tuples.
2	len(tuple)  Gives the total length of the tuple.
3	max(tuple)  Returns item from the tuple with max value.
4	min(tuple)  Returns item from the tuple with min value.
5	tuple(seq)  Converts a list into tuple.

Lists

Lists are variable-length and their contents can be modified in-place. You can define them using square brackets `[]` or using the `list` type function:

```
In [56]: a_list = [2, 3, 7, None]
         tup = ('foo', 'bar', 'baz')
```

```
In [57]: b_list = list(tup)
```

```
In [58]: b_list
```

```
Out[58]: ['foo', 'bar', 'baz']
```

```
In [59]: b_list[1] = 'peekaboo'
```

```
In [61]: b_list
```

```
Out[61]: ['foo', 'peekaboo', 'baz']
```

The `list` function is frequently used in data processing as a way to materialize an iterator or generator expression:

```
In [65]: gen = range(10)
```

```
In [67]: list(gen)
```

```
Out[67]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Adding and removing elements

Elements can be appended to the end of the list with the **append** method:

```
In [70]: b_list.append('dwarf')
```

```
In [71]: b_list
```

```
Out[71]: ['foo', 'peekaboo', 'baz', 'dwarf', 'dwarf', 'dwarf']
```

Using **insert** you can insert an element at a specific location in the list:

```
In [74]: b_list.insert(1, 'red')
```

```
In [77]: b_list
```

```
Out[77]: ['foo', 'red', 'red', 'peekaboo', 'baz', 'dwarf', 'dwarf', 'dwarf']
```

The inverse operation to insert is **pop**, which removes and returns an element at a particular index:

```
In [79]: b_list.pop(2)
```

```
Out[79]: 'peekaboo'
```

```
In [83]: b_list
```

```
Out[83]: ['red', 'baz', 'dwarf', 'dwarf', 'dwarf']
```

Elements can be removed by value with **remove**, which locates the first such value and removes it from the list:

```
In [86]: b_list.remove('red')
```

```
In [88]: b_list
```

```
Out[88]: ['baz', 'dwarf', 'dwarf', 'dwarf']
```

Check if a list contains a value using the **in / not in** keyword:

```
In [90]: 'dwarf' in b_list
```

```
Out[90]: True
```

```
In [91]: 'dwarf' not in b_list
```

```
Out[91]: False
```

Concatenating and combining lists

```
In [92]: [4, None, 'foo'] + [7, 8, (2, 3)]
```

```
Out[92]: [4, None, 'foo', 7, 8, (2, 3)]
```

If you have a list already defined, you can append multiple elements to it using the **extend** method:

```
In [94]: x = [4, None, 'foo']
```

```
In [96]: x.extend([7, 8, (2, 3)])
```

```
In [97]: x
```

```
Out[97]: [4, None, 'foo', 7, 8, (2, 3), 7, 8, (2, 3)]
```

Note that list concatenation by addition is a comparatively expensive operation since a new list must be created and the objects copied over. Using extend to append elements to an existing list, especially if you are building up a large list, is usually preferable.

```
In [ ]: """
everything = []
for chunk in list_of_lists:
    everything.extend(chunk)

#is faster than

everything = []
for chunk in list_of_lists:
    everything = everything + chunk
"""
```

Sorting

```
In [3]: data = [7, 2, 5, 1, 3]
```

Sort and creating a new object using **sorted()**

```
In [5]: data2 = sorted(data) #perform COPIED sorting on the data
```

```
In [6]: data
```

```
Out[6]: [7, 2, 5, 1, 3]
```

```
In [7]: data2
```

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

You can sort a list in-place (without creating a new object) by calling its **sort** function:

```
In [8]: data.sort()
```

```
In [10]: data
```

```
Out[10]: [1, 2, 3, 5, 7]
```

One is the ability to pass a secondary sort key—that is, a function that produces a value to use to sort the objects. For example, we could sort a collection of strings by their lengths:

```
In [12]: b = ['saw', 'small', 'He', 'foxes', 'six']
```

```
In [13]: b.sort(key=len)
```

```
In [14]: b
```

```
Out[14]: ['He', 'saw', 'six', 'small', 'foxes']
```

Binary search and maintaining a sorted list

```
In [17]: import bisect
```

The built-in **bisect** module implements binary search and insertion into a **sorted list**. **bisect.bisect** finds the location where an element should be inserted to keep it sorted, while **bisect.insort** actually inserts the element into that location:

```
In [18]: c = [1, 2, 2, 2, 3, 4, 7]
```



```
In [19]: bisect.bisect(c, 2)
```

```
Out[19]: 4
```

```
In [21]: bisect.bisect(c, 5)
```

```
Out[21]: 6
```

```
In [22]: bisect.insort(c, 6)
```

```
In [23]: c
```

```
Out[23]: [1, 2, 2, 2, 3, 4, 6, 7]
```

Slicing

You can select sections of most sequence types by using slice notation, which in its basic form consists of *start:stop* passed to the indexing operator `[]`:

```
In [25]: seq = [7, 2, 3, 7, 5, 6, 0, 1]
         seq[1:5]
```

```
Out[25]: [2, 3, 7, 5]
```

Slices can also be assigned to with a sequence:

```
In [26]: seq[3:4] = [6, 3]
         seq
```

```
Out[26]: [7, 2, 3, 6, 3, 5, 6, 0, 1]
```

While the element at the *start* index is *included*, the *stop* index is **not included**, so that the number of elements in the result is *stop - start*.

Either the start or stop can be omitted, in which case they default to the start of the sequence and the end of the sequence, respectively:

```
In [27]: seq[:5]
```

```
Out[27]: [7, 2, 3, 6, 3]
```

```
In [28]: seq[3:]
```

```
Out[28]: [6, 3, 5, 6, 0, 1]
```

Negative indices slice the sequence relative to the end:

```
In [30]: seq[-4:]
```

```
Out[30]: [5, 6, 0, 1]
```

```
In [31]: seq[-6:-2]
```

```
Out[31]: [6, 3, 5, 6]
```

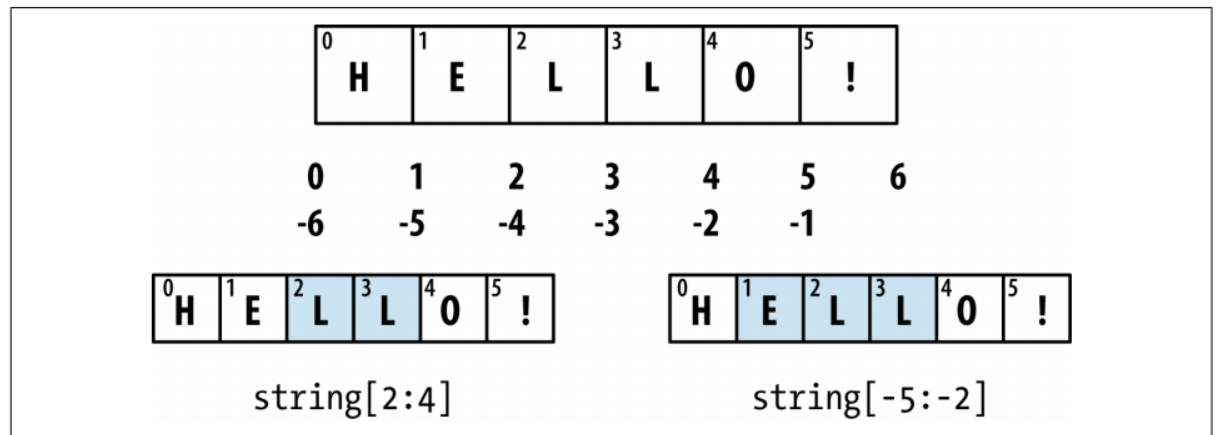


Illustration of Python slicing conventions

A **step** can also be used after a second colon to, say, take every other element:

```
In [35]: seq[::2]
```

```
Out[35]: [7, 3, 3, 6, 1]
```

A clever use of this is to pass -1, which has the useful effect of reversing a list or tuple:

```
In [36]: seq[::-1]
```

```
Out[36]: [1, 0, 6, 5, 3, 6, 3, 2, 7]
```

Built-in Sequence Functions

enumerate

Returns a sequence of (i, value) tuples:

```
In [39]: some_list = ['foo', 'bar', 'baz']
mapping = {}
```

```
In [40]: for i, v in enumerate(some_list):  
         mapping[v] = i
```

```
In [41]: mapping
```

```
Out[41]: {'bar': 1, 'baz': 2, 'foo': 0}
```

sorted

Returns a *new* sorted list from the elements of any sequence:

```
In [42]: sorted([7, 1, 2, 6, 0, 3, 2])
```

```
Out[42]: [0, 1, 2, 2, 3, 6, 7]
```

```
In [43]: sorted('horse race')
```

```
Out[43]: [' ', 'a', 'c', 'e', 'e', 'h', 'o', 'r', 'r', 's']
```

zip

“pairs” up the elements of a number of lists, tuples, or other sequences to create a list of tuples:

```
In [45]: seq1 = ['foo', 'bar', 'baz']  
         seq2 = ['one', 'two', 'three']  
         seq3 = [False, True]
```

```
In [52]: zipped = zip(seq1, seq2)
```

```
In [53]: zipped
```

```
Out[53]: <zip at 0x59e7908>
```

```
In [54]: list(zipped)
```

```
Out[54]: [('foo', 'one'), ('bar', 'two'), ('baz', 'three')]
```

zip can take an arbitrary number of sequences, and the number of elements it produces is determined by the *shortest* sequence:

```
In [55]: list(zip(seq1, seq2, seq3))
```

```
Out[55]: [('foo', 'one', False), ('bar', 'two', True)]
```

A very common use of `zip` is simultaneously iterating over multiple sequences, possibly also combined with `enumerate`:

```
In [56]: for i, (a, b) in enumerate(zip(seq1, seq2)):
         print('{0}: {1}, {2}'.format(i, a, b))
```

```
0: foo, one
1: bar, two
2: baz, three
```

Given a “zipped” sequence, `zip` can be applied in a clever way to “unzip” the sequence. Another way to think about this is converting a list of rows into a list of columns. The syntax, which looks a bit magical, is:

```
In [57]: pitchers = [('Nolan', 'Ryan'), ('Roger', 'Clemens'), ('Schilling', 'Curt')]
         first_names, last_names = zip(*pitchers)
```

```
In [58]: first_names
```

```
Out[58]: ('Nolan', 'Roger', 'Schilling')
```

```
In [60]: last_names
```

```
Out[60]: ('Ryan', 'Clemens', 'Curt')
```

reversed

iterates over the elements of a sequence in reverse order:

```
In [62]: list(reversed(range(10)))
```










```
Out[62]: [9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

Keep in mind that `reversed` is a generator, so it does not create the reversed sequence until materialized (e.g., with `list` or a `for` loop).

```
In [3]: list = [ 'abcd', 786 , 2.23, 'john', 70.2 ]
        tinylist = [123, 'john']






        print (list)           # Prints complete List
        print (list[0])        # Prints first element of the List
        print (list[1:3])      # Prints elements starting from 2nd till 3rd
        print (list[2:])       # Prints elements starting from 3rd element
        print (tinylist * 2)    # Prints List two times
        print (list + tinylist) # Prints concatenated Lists

['abcd', 786, 2.23, 'john', 70.2]
abcd
[786, 2.23]
[2.23, 'john', 70.2]
[123, 'john', 123, 'john']
['abcd', 786, 2.23, 'john', 70.2, 123, 'john']
```

Sr.No.	Methods with Description
1	list.append(obj)  Appends object obj to list
2	list.count(obj)  Returns count of how many times obj occurs in list
3	list.extend(seq)  Appends the contents of seq to list
4	list.index(obj)  Returns the lowest index in list that obj appears
5	list.insert(index, obj)  Inserts object obj into list at offset index
6	list.pop(obj=list[-1])  Removes and returns last object or obj from list
7	list.remove(obj)  Removes object obj from list
8	list.reverse()  Reverses objects of list in place
9	list.sort([func])  Sorts objects of list, use compare func if given

Built-in List Functions & Methods

Python includes the following list functions –

Sr.No.	Function with Description
1	cmp(list1, list2)  Compares elements of both lists.
2	len(list)  Gives the total length of the list.
3	max(list)  Returns item from the list with max value.
4	min(list)  Returns item from the list with min value.
5	list(seq)  Converts a tuple into list.

Dictionaries

It is a flexibly sized collection of key-value pairs, where key and value are Python objects. One approach for creating one is to use curly braces { } and colons to separate keys and values:

```
In [64]: empty_dict = {}
```

```
In [65]: d1 = {'a' : 'some value', 'b' : [1, 2, 3, 4]}
```

```
In [66]: d1
```

```
Out[66]: {'a': 'some value', 'b': [1, 2, 3, 4]}
```

You can access, insert, or set elements using the same syntax as for accessing elements of a list or tuple:

```
In [67]: d1[7] = 'an integer'
```

```
In [68]: d1
```

```
Out[68]: {7: 'an integer', 'a': 'some value', 'b': [1, 2, 3, 4]}
```

```
In [69]: d1['b']
```

```
Out[69]: [1, 2, 3, 4]
```

You can check if a dict contains a key using the same syntax used for checking whether a list or tuple contains a value:

```
In [71]: 'b' in d1
```

```
Out[71]: True
```

You can delete values either using the **del** keyword or the **pop** method (which simultaneously returns the value and deletes the key):

```
In [73]: d1[5] = 'some value'
```

```
In [74]: d1
```

```
Out[74]: {5: 'some value', 7: 'an integer', 'a': 'some value', 'b': [1, 2, 3, 4]}
```

```
In [76]: d1['dummy'] = 'another value'
```

```
In [77]: d1
```

```
Out[77]: {5: 'some value',  
          7: 'an integer',  
          'a': 'some value',  
          'b': [1, 2, 3, 4],  
          'dummy': 'another value'}
```

```
In [78]: del d1[5]
```

```
In [79]: d1
```

```
Out[79]: {7: 'an integer',  
          'a': 'some value',  
          'b': [1, 2, 3, 4],  
          'dummy': 'another value'}
```

```
In [80]: ret = d1.pop('dummy')
```



```
In [81]: ret
```

```
Out[81]: 'another value'
```

```
In [82]: d1
```

```
Out[82]: {7: 'an integer', 'a': 'some value', 'b': [1, 2, 3, 4]}
```

The **keys** and **values** method give you iterators of the dict's keys and values, respectively. While the key-value pairs are not in any particular order, these functions output the keys and values in the same order:

```
In [83]: list(d1.keys())
```

```
Out[83]: ['a', 'b', 7]
```

```
In [84]: list(d1.values())
```

```
Out[84]: ['some value', [1, 2, 3, 4], 'an integer']
```

You can merge one dict into another using the **update** method:

```
In [85]: d1.update({'b' : 'foo', 'c' : 12})
```

```
In [86]: d1
```

```
Out[86]: {7: 'an integer', 'a': 'some value', 'b': 'foo', 'c': 12}
```

The **update** method changes dicts *in-place*, so any existing keys in the data passed to update will have their old values discarded.

Creating dicts from sequences

```
In [87]: """  
mapping = {}  
for key, value in zip(key_list, value_list):  
    mapping[key] = value  
"""
```

```
Out[87]: '\nmapping = {}\nfor key, value in zip(key_list, value_list):\n    mapping[key] = value\n'
```

```
In [88]: mapping = dict(zip(range(5), reversed(range(5))))
```

```
In [89]: mapping
```

```
Out[89]: {0: 4, 1: 3, 2: 2, 3: 1, 4: 0}
```

Default values

The dict methods **get** and **pop** can take a *default* value to be returned:

```
In [92]: """value = some_dict.get(key, default_value)"""
```

```
Out[92]: 'value = some_dict.get(key, default_value)'
```

get by default will return **None** if the key is not present, while **pop** will raise an exception.

With setting values, a common case is for the values in a dict to be other collections, like lists. For example, you could imagine categorizing a list of words by their first letters as a dict of lists:

```
In [93]: words = ['apple', 'bat', 'bar', 'atom', 'book']
```

```
In [94]: by_letter = {}
```

```
In [95]: for word in words:
         letter = word[0]
         if letter not in by_letter:
             by_letter[letter] = [word]
         else:
             by_letter[letter].append(word)
```

```
In [96]: by_letter
```

```
Out[96]: {'a': ['apple', 'atom'], 'b': ['bat', 'bar', 'book']}
```

The **setdefault** dict method is for precisely this purpose. The preceding for loop can be rewritten as:

```
In [98]: for word in words:
         letter = word[0]
         by_letter.setdefault(letter, []).append(word)
```

The built-in *collections* module has a useful class, **defaultdict**, which makes this even easier. To create one, you pass a type or function for generating the default value for each slot in the dict:

```
In [100]: from collections import defaultdict
```

```
In [101]: by_letter = defaultdict(list)
```

```
In [102]: by_letter
```

```
Out[102]: defaultdict(list, {})
```

```
In [103]: for word in words:
           by_letter[word[0]].append(word)
```

```
In [104]: by_letter
```

```
Out[104]: defaultdict(list, {'a': ['apple', 'atom'], 'b': ['bat', 'bar', 'book']})
```

Valid dict key types

While the **values of a dict can be any Python object**, the **keys generally have to be immutable objects** like scalar types (int, float, string) or tuples (all the objects in the tuple need to be immutable, too). The technical term here is **hashability**. You can check whether an object is hashable (can be used as a key in a dict) with the hash function:

```
In [105]: hash('string')
```

```
Out[105]: 6516797597419965410
```

```
In [106]: hash((1, 2, (2, 3)))
```

```
Out[106]: 1097636502276347782
```

```
In [107]: hash((1, 2, [2, 3]))
```

```
-----
TypeError                                Traceback (most recent call last)
<ipython-input-107-8ffc25aff872> in <module>()
----> 1 hash((1, 2, [2, 3]))
```

```
TypeError: unhashable type: 'list'
```

To use a list as a key, one option is to convert it to a tuple, which can be hashed as long as its elements also can:

```
In [108]: d = {}
```

```
In [109]: d[tuple([1, 2, 3])] = 5
```

In [7]: d





```
-----  
NameError                                Traceback (most recent call last)  
<ipython-input-7-e983f374794d> in <module>()  
----> 1 d  
  
NameError: name 'd' is not defined
```








```
In [9]: dict = {}  
dict['one'] = "This is one"  
dict[2]     = "This is two"  
  
tinydict = {'name': 'john', 'code': 6734, 'dept': 'sales'}  
  
print (dict['one'])      # Prints value for 'one' key  
print (dict[2])         # Prints value for 2 key  
print (tinydict)        # Prints complete dictionary  
print (tinydict.keys()) # Prints all the keys  
print (tinydict.values()) # Prints all the values
```




```
This is one  
This is two  
{'name': 'john', 'code': 6734, 'dept': 'sales'}  
dict_keys(['name', 'code', 'dept'])  
dict_values(['john', 6734, 'sales'])
```

Built-in Dictionary Functions & Methods

Python includes the following dictionary functions –

Sr.No.	Function with Description
1	cmp(dict1, dict2)  Compares elements of both dict.
2	len(dict)  Gives the total length of the dictionary. This would be equal to the number of items in the dictionary.
3	str(dict)  Produces a printable string representation of a dictionary
4	type(variable)  Returns the type of the passed variable. If passed variable is dictionary, then it would return a dictionary type.

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

8	<code>dict.setdefault(key, default=None)</code>  Similar to <code>get()</code> , but will set <code>dict[key]=default</code> if <i>key</i> is not already in dict
9	<code>dict.update(dict2)</code>  Adds dictionary <i>dict2</i> 's key-values pairs to <i>dict</i>
10	<code>dict.values()</code>  Returns list of dictionary <i>dict</i> 's values

sets

A **set** is an unordered collection of *unique* elements. You can think of them like dicts, but keys only, no values. A set can be created in two ways: via the **set** function or via a set *literal* with curly braces:

```
In [111]: set([2, 2, 2, 1, 3, 3])
```

```
Out[111]: {1, 2, 3}
```

```
In [112]: {2, 2, 2, 1, 3, 3}
```

```
Out[112]: {1, 2, 3}
```

Sets support mathematical set operations like *union*, *intersection*, *difference*, and *symmetric difference*.

```
In [114]: a = {1, 2, 3, 4, 5}
```

```
In [115]: b = {3, 4, 5, 6, 7, 8}
```

```
In [116]: a.union(b)
```

```
Out[116]: {1, 2, 3, 4, 5, 6, 7, 8}
```

```
In [117]: a | b
```

```
Out[117]: {1, 2, 3, 4, 5, 6, 7, 8}
```

```
In [118]: a.intersection(b)
```

```
Out[118]: {3, 4, 5}
```

```
In [119]: a & b
```

```
Out[119]: {3, 4, 5}
```

![alt text](images/set operations.png "Set operations")

All of the logical set operations have in-place counterparts, which enable you to replace the contents of the set on the left side of the operation with the result. For very large sets, this may be more efficient:

```
In [120]: c = a.copy()
```

```
In [121]: c
```

```
Out[121]: {1, 2, 3, 4, 5}
```

```
In [122]: c |= b
```

```
In [123]: c
```

```
Out[123]: {1, 2, 3, 4, 5, 6, 7, 8}
```

```
In [124]: d = a.copy()
```

```
In [125]: d &= b
```

```
In [126]: d
```

```
Out[126]: {3, 4, 5}
```

Like dicts, set elements generally must be immutable. To have list-like elements, you must convert it to a tuple:

```
In [128]: my_data = [1, 2, 3, 4]
```

```
In [129]: my_set = {tuple(my_data)}
```

```
In [130]: my_set
```

```
Out[130]: {(1, 2, 3, 4)}
```

You can also check if a set is a subset of or a superset of another set:

```
In [132]: a_set = {1, 2, 3, 4, 5}
```

```
In [133]: {1, 2, 3}.issubset(a_set)
```

```
Out[133]: True
```

```
In [134]: a_set.issuperset({1, 2, 3})
```

```
Out[134]: True
```

Sets are equal if and only if their contents are equal:

```
In [135]: {1, 2, 3} == {3, 2, 1}
```

```
Out[135]: True
```

List, Set, and Dict Comprehensions

General form:

[expr for val in collection if condition]

```
In [137]: """
[expr for val in collection if condition]

This is equivalent to the following for loop:

result = []
for val in collection:
    if condition:
        result.append(expr)
"""
```

```
Out[137]: '\n[expr for val in collection if condition]\n\nThis is equivalent to the following for loop:\n\nresult = []\nfor val in collection:\n    if condition:\n        result.append(expr)\n'
```

```
In [138]: strings = ['a', 'as', 'bat', 'car', 'dove', 'python']
```



```
In [139]: [x.upper() for x in strings if len(x) > 2]
```

```
Out[139]: ['BAT', 'CAR', 'DOVE', 'PYTHON']
```

Set and dict comprehensions:

```
In [141]: """
dict_comp =
{key-expr : value-expr for value in collection if condition}
"""
```

```
Out[141]: '\ndict_comp = \n{key-expr : value-expr for value in collection if condition}
\n'
```

```
In [142]: """set_comp = {expr for value in collection if condition}"""
```

```
Out[142]: 'set_comp = {expr for value in collection if condition}'
```

```
In [143]: unique_lengths = {len(x) for x in strings}
```

```
In [144]: unique_lengths
```

```
Out[144]: {1, 2, 3, 4, 6}
```

We could also express this more functionally using the **map** function, introduced shortly:

```
In [146]: set(map(len, strings))
```

```
Out[146]: {1, 2, 3, 4, 6}
```

```
In [147]: loc_mapping = {val : index for index, val in enumerate(strings)}
```

```
In [148]: loc_mapping
```

```
Out[148]: {'a': 0, 'as': 1, 'bat': 2, 'car': 3, 'dove': 4, 'python': 5}
```

Nested list comprehensions

```
In [149]: all_data = [['John', 'Emily', 'Michael', 'Mary', 'Steven'], ['Maria', 'Juan', 'Ja
```

Suppose we wanted to get a single list containing all names with two or more e's in them. We could certainly do this with a simple for loop:

```
In [150]: names_of_interest = []
         for names in all_data:
             enough_es = [name for name in names if name.count('e') >= 2]
             names_of_interest.extend(enough_es)
```

```
In [152]: names_of_interest
```

```
Out[152]: ['Steven']
```

You can actually wrap this whole operation up in a single nested list comprehension:

```
In [153]: result = [name for names in all_data
                    for name in names
                    if name.count('e') >= 2]
```

```
In [154]: result
```

```
Out[154]: ['Steven']
```

```
In [155]: some_tuples = [(1, 2, 3), (4, 5, 6), (7, 8, 9)]
```

```
In [156]: flattened = [x for tup in some_tuples for x in tup]
```

```
In [157]: flattened
```

```
Out[157]: [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
In [158]: """
         flattened = []
         for tup in some_tuples:
             for x in tup:
                 flattened.append(x)
         """
```

```
Out[158]: '\nflattened = []\nfor tup in some_tuples:\n    for x in tup:\n        flattene
         d.append(x)\n'
```

```
In [159]: [[x for x in tup] for tup in some_tuples]
```

```
Out[159]: [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

```
In [ ]:
```

