Would you like to see my collection?
A quick introduction

Brian Hare
January 2015

This is NOT an exhaustive listing of everything you can do with these, just an overview for the beginner to get you started. Play with it! Python encourages exploration.

There are 4 basic collections in Python:

- Lists
- Tuples
- Sets
- Dictionaries

Lists and *Tuples* are ordered; you can refer to individual items by their position in the list. Indexing starts at 0.

```
>>> L = [3, 2, 1]
>>> for j in range(len(L)):
      print("Position", j, "contains", L[j])


Position 0 contains 3
Position 1 contains 2
Position 2 contains 1
>>>
```

The len() function returns the number of items in a collection and works for any collection.

The primary difference between lists and tuples is that lists are mutable (can be changed), while tuples are immutable (cannot be changed).

```
>>> L = [1, 2, 3]
>>> T = ('a', 'b', 'c')
>>> L.append(4)
>>> L
[1, 2, 3, 4]
>>> T.append('d')
Traceback (most recent call last):
  File "<pyshell#8>", line 1, in <module>
    T.append('d')
AttributeError: 'tuple' object has no attribute 'append'
>>>
```

Notice that lists are denoted using square brackets and tuples by parentheses.

The following method are defined for tuples and lists:

Operation	Result
`x in s`	`True` if an item of *s* is equal to *x*, else `False`
`x not in s`	`False` if an item of *s* is equal to *x*, else `True`
`s + t`	the concatenation of *s* and *t*
`s * n` or `n * s`	*n* shallow copies of *s* concatenated
`s[i]`	*i*th item of *s*, origin 0
`s[i:j]`	slice of *s* from *i* to *j*
`s[i:j:k]`	slice of *s* from *i* to *j* with step *k*
`len(s)`	length of *s*
`min(s)`	smallest item of *s*
`max(s)`	largest item of *s*
`s.index(x[, i[, j]])`	index of the first occurrence of *x* in *s* (at or after index *i* and before index*j*)
`s.count(x)`	total number of occurrences of *x* in *s*
`sorted(s)`	Returns a new list with the elements of s, sorted
`C == D`	True if C and D are the same type and have the same items in the same order, False otherwise
`C != D`	False if C and D are the same type and have the same items in the same order, True otherwise
`C is D`	True if C and D are aliases for the same object, False otherwise

(source: http://docs.python.org)

For comparisons other than equality, lists and tuples can be compared to other items of the same type (lists can be compared to lists, tuples to tuples) using the normal comparison operators; comparison is element-by-element, until the first difference is encountered.

```
>>> L1 = [1, 2, 3, 3]
>>> L2 = [1, 2, 4, 3]
>>> L2 > L1
True
```

The following methods are defined for lists but not for tuples:

Operation	Result
`s[i] = x`	item *i* of *s* is replaced by *x*
`s[i:j] = t`	slice of *s* from *i* to *j* is replaced by the contents of the iterable *t*
`del s[i:j]`	same as `s[i:j] = []`
`s[i:j:k] = t`	the elements of `s[i:j:k]` are replaced by those of *t*
`del s[i:j:k]`	removes the elements of `s[i:j:k]` from the list
`s.append(x)`	appends *x* to the end of the sequence (same as `s[len(s):len(s)] = [x]`)
`s.clear()`	removes all items from `s` (same as `del s[:]`)
`s.copy()`	creates a shallow copy of `s` (same as `s[:]`)

Operation	Result
`s.extend(t)`	extends *s* with the contents of *t* (same as `s[len(s):len(s)] =t`)
`s.insert(i, x)`	inserts *x* into *s* at the index given by *i* (same as `s[i:i] = [x]`)
`s.pop([i])`	retrieves the item at *i* and also removes it from *s*
`s.remove(x)`	remove the first item from *s* where `s[i] == x`
`s.reverse()`	reverses the items of *s* in place

Here's a potential gotcha:
Suppose you have a list that you need to filter, removing certain items as you go. You may be tempted to do this:

```
#DON'T DO IT THIS WAY
for item in L:
    if BadItem(item):
         L.remove(item)
```

The problem is that you're changing the length of the list while iterating through it, which will cause problems before the loop is finished.

```
# DO THIS INSTEAD
M = list() # or M = []
for item in L:
    if not BadItem(item):
         M.append(item)
L = M # keep only the items that weren't bad
```

List comprehensions (a topic for another day) are very useful for this kind of filtering operation.

Here's another gotcha:

The sorted() function returns a new, sorted list, leaving the original list unchanged. The list's .sort() method sorts the list in place, returning nothing. Sooner or later you may do something like this:

```
>>> L = [6, 4, 2]
>>> L = L.sort()  # DANGER, WILL ROBINSON!
>>> L  # where did it go?
>>>
>>> type(L)
<class 'NoneType'>
>>>
```

What happened? Well, L.sort() sorted L in place, and didn't return anything. When we try to assign that nothing back to L, what gets assigned is the special value None. *The original list was sorted, then discarded.* Too bad if that was data you needed. L.sort() and sorted(L) are both useful, but are not interchangeable, and you should be very careful when mixing them. Make sure you're clear on what each does. In the line above, `L = sorted(L)` would have been fine (though calling L.sort() is a little faster).

```
L.sort()   # Sorts L in place, returns None
sorted(L)  # Returns a new list with L's values sorted, L is unchanged
```

Sets and *Dictionaries* are unordered; you cannot use index numbers to refer to individual elements.

Sets are unordered collections; like lists and tuples, they can be heterogeneous (containing different data types).

```
>>> S = set()
>>> S.add(5)
>>> S.add('a')
>>> S.add(3.41)
>>> S
{3.41, 5, 'a'}
>>> S = set(L1)
>>> S
{1, 2, 3}
>>> S[0]
Traceback (most recent call last):
  File "<pyshell#35>", line 1, in <module>
    S[0]
TypeError: 'set' object does not support indexing
```

Set operations supported:

Operation	Result
`S.union(T)` `S | T`	Set union – anything in S or in T
`S.intersection(T)` `S & T`	Set intersection – only those elements in both S and T
`len(s)`	How many elements in s
`s.issubset(t)` `s <= t`	Tests whether s is a subset of t
`S < T`	Tests whether S is a proper subset of T
`s.issuperset(t)` `S >= T`	Tests whether S is a superset of T
`S > T`	Tests whether S is a proper superset of T
`S == T, S != T`	Tests whether sets are equal
`S - T`	Returns new set, elements of S that are not in T (S-T != T-S)
`s.symmetric_difference(t)` `S ^ T`	Returns new set, elements that are in S or T but not both
`S += T`	Add elements of T to S
`S -= T`	Remove elements of T from S
`S ^= T`	Keep elements in S or T but not both

Operation	Result
`s.add(item)`	Put new element into set
`s.remove(item)`	Remove item from set, raise error if it's not present
`s.discard(item)`	Remove element from set, ignore it if not present
`s.pop()`	Return (and remove) an arbitrary item from set
`s.clear()`	Remove everything from set

Sets do not allow duplicate items. Adding an item to a set that's already in the set has no effect. This leads to a fast way to remove duplicate items from a list:

```
>>> L = ['apam', 'spam', 'spam', 'spam', 'baked beans', 'fried egg',
'spam']
>>> Uniques = list(set(L))
>>> Uniques
['spam', 'apam', 'fried egg', 'baked beans']
>>>
```

This does destroy any ordering in the original list; if that's relevant, sets shouldn't be used.

Sets are mutable. For reasons we're about to see, sometimes we want something that's like a set but is immutable. Python has a `frozenset` type:

```
>>> F = frozenset(Uniques)
>>> F
frozenset({'spam', 'apam', 'fried egg', 'baked beans'})
>>>
```

A frozenset has all of the functionality above that returns new sets, and can test for membership and comparison, but does not have add(), remove(), discard(), pop(), or any other method that changes the membership of the set.

Dictionaries are another unordered collection. A dictionary stores key-value pairs. Keys must be immutable: numbers, strings, tuples, etc. Sets are mutable and so cannot be dictionary keys, but frozensets are immutable and can be keys. (That's the main reason they're provided, in fact.) Values can be anything, mutable or not. Neither keys nor values need to be homogeneous.

```
>>> D = dict()
>>> D['Expected'] = ['Live parrot'] # value is a list holding a string
>>> D['Unexpected'] = ['Dead parrot']
>>> D['Unexpected'].append('Spanish Inquisition')
>>> D['Unexpected']
['Dead parrot', 'Spanish Inquisition']
>>>
```

Dictionaries support the following operations:

Operation	Result
`len(D)`	Returns number of items (key/value pairs) in D

Operation	Result
`D[k]`	Returns item associated with key k, KeyError if not present
`k in D`	True if k is a key in D, False otherwise
`D.clear()`	Removes all items from D
`D.keys()`	Returns an iterable collection of keys from D
`D.values()`	Returns an iterable collection of values from D
`D.items()`	Returns an iterable collection of (key, value) tuples from D
`D.popitem()`	Return (and remove from D) an arbitrary item from D
`D.update(E)`	Add keys/values of dictionary E to D. If same key is in both, D's key is overwritten
`for k in D:`	Iterate through keys in D
`for k in sorted(D):`	Iterate through keys in order (keys must be comparable)
`del D[k]`	Remove k and its value from D. Error if k is not a key.