Python Tuples

The "tuple" is a Python type which is like a small list. Tuples are like a list but written within parenthesis, here is a tuple of three strings: `('a', 'b', 'c')`

Accessing a tuple works like a list — len(), [ ], in, slices, loops — but the big difference is that a tuple is immutable. It is built once and then never changes. There is no append() function.

```
>>> t = ('a', 'b', 'c')
>>> len(t)
3
>>> t[0]
'a'
>>> 'c' in t
True
>>> t[0:2]
('a', 'b')
>>> t[0:2]
('a', 'b')
>>> t[0] = 'hi'   # no changing it!
Error:'tuple' object does not support item assignment
```

Tuples are useful where you have a fixed number of things to keep as a group — e.g. an x,y coordinate pair stored in a len-2 tuple like (4, 10) keeps the x,y together as a unit. In contrast, a list is useful when you have an unlimited number of items you want to store together, such as the typical pattern of reading out of a file and using lst.append() to put all the items together in one list.

Recall that dict-keys should be immutable. Therefore if you have some composite data that you want to use as a dict key, e.g. a string and an int, form them into a tuple ('meh', 16) and then use that as the key.

Tuples and Assignment

One quirky but important use of tuples is assigning multiple variable at once, like this:

```
>>> (x, y) = (12, 4)
>>> x
12
>>> y
4
```

The length of the left-hand-side and right-hand-side must be the same (CS terminology protip the "lhs" and "rhs".)

Tuple Syntax Option

It's possible to write a tuple in Python without the parenthesis. The presence of the comma is enough to cue Python that this is a tuple.

```
>>> t1 = (1, 2, 3)
>>> t1
(1, 2, 3)
>>> t2 = 1, 2, 3    # works without parens
>>> t2
(1, 2, 3)
```

As a matter of style, we prefer to write tuples with the parenthesis. In the spirit of readability, we want the look of the code to reflect what is going on, and it's not like the parenthesis are a big visual distraction.

Tuples and Multiple Value Return

Communicating **in** to a function is a rich area: you can have any number of parameters, they each get a name. How do you communicate **out** of a function .. return 1 value. Returning 1 value covers 90% of the cases. But sometimes it really makes sense to return 2 or more values. The Pythonic way to do this is to return a tuple packing together the values. Like this

```
def min2(a, b):
    """
    Given 2 ints, returns (True, min_val) if at least
    one is negative and min_val is the minimum value.
    Otherwise returns (False, None)
    """
    if a < 0 or b < 0:
        return (True, min(a, b))
    return (False, None)
```

There are 2 requirements:

1. The function doc string must state the size and content-types of the returned tuple. There's no way the caller code can be written correctly without this information.

2. All paths should return a tuple of that same length, even if it is (None, None), so that standard looking calling code of the form (x, y) = fn(..) works without crashing should it hit the None case.

Calling min2 looks like this, catching the len-2 tuple result into 2 variables.

```
(negative, min_val) = min2(a, b)
if negative:
    ...
```

Tuples and Dicts

One handy use of tuples is the dict.items() function, which returns the entire contents of the dict as a list of len-2 (key, value) tuples.

```
>>> d = {'a':1, 'd':4, 'c':2, 'b':3}
>>> d
{'a': 1, 'd': 4, 'c': 2, 'b': 3}
>>> d.items()
dict_items([('a', 1), ('d', 4), ('c', 2), ('b', 3)])
# (key, value) tuples
>>> sorted(d.items())   # same tuples, sorted by key
[('a', 1), ('b', 3), ('c', 2), ('d', 4)]
```

Sorting of tuples goes left-to-right within each tuple - first sorting by all the [0] values across the tuples, then by [1], and so on.

Once all the data has been loaded into a dict, it's natural to have a process-all-the-data phase. This can be written as a loop over the above d.items() list. The loop syntax below takes one tuple off the list for each iteration, setting the two variable, key and value each time:

```
# Example: for loop setting key/value for each
iteration
for key,value in d.items():
    # use key and value in here
```

This is a special version of the for loop, where there are multiple variables, and the number of variables matches the size of a tuples coming off the list. The above example, looping key,value over dict.items() is probably the most common use of this multiple-variable variant of the for loop.