6.00.1x Week 3 FAQ

- **What is the difference between a tuple and a list?**
  They are *very* similar. The notation is different. A tuple is specified as a comma-deliminated list of values surrounded by parentheses. For example `t = (1, 'one')`. The important thing to remember is that if you only have one element in a tuple **you must follow it with a comma**. For example `t = ('one' , )`. A tuple is immutable, meaning you can't change an element of a tuple. You can only replace a tuple completely by assigning a whole new tuple to the variable. For example if you tried to code `t[0] = 2` you would get an error, but you *could* say `t = (2, 'two')`.

  A list is specified with a "list" of values surrounded by square brackets.
  For example `l = [1, 'one']`. If you have only one element you do *not* want a comma after it (i.e. `l = ['one']`).

- **Can you use slicing with a tuple or a list?**
  Yes. You code it the same way you would slice a string.
  For example if you had `t = (1, 'one')` then `t[1:2]` would give you a tuple equal to `('one' ,)`. Slicing a list would give you another list.

- **Does indexing a tuple return an element of type tuple?**
  No. If you have `t = (2, "one", 3)` then `t[0]` will return 2. If you want a tuple you need to code a slice (i.e. `t[ : 1]`). Per the reply to this question.

- **Can tuples and lists contain tuples and lists within them?**
  Yes. For example you can have `t = (1, 2, ('one', 'two'))`. To access the 'two' you would need index [2] of `t`, then within that you would need index 1 of the inside tuple. In other words `t[2][1] == 'two'` is True.

- **What is the difference between list.append and list.extend?**
  List.*append* takes whatever the item is and adds it to the end of the list. The item can be an int, a string, another list or a tuple or anything. For example if `l = [1, 2]` then `l.append([3, 4])` will give you `l = [1, 2, [3, 4]]`. With list.*extend* the item needs to be a kind of "iterable" like a string, tuple or another list. Each element of the list is individually added to the end of the first list. For example if `l = [1, 2]` then `l.extend([ (3,), 4 ])` will give you `l = [1, 2, (3,), 4]`.

- **Where do I get information on all the functions that apply to lists?**
  The best information source is the Python docs.

- **What do the various list functions return?**
  The list functions don't return anything. They merely change the value of the list being operated on.

- **What is the difference between list.sort and sorted(list)?**
  Using `list.sort` changes the value of the variable `list`. The function `sorted(list)` returns a new variable that is sorted. The code would be `listA = sorted(listB)`.

Listing 1: Difference between list.sort and sorted(list)

```
>>> listA = ['a', 'c', 'b']
>>> listB = ['b', 'a', 'c']
>>>
>>> # listA.sort() change the value of listA
... listA.sort()
>>>
>>> # sorted(listB) return a new variable
... listB = sorted(listB)
>>>
>>> listA
['a', 'b', 'c']
>>> listB
['a', 'b', 'c']
```

- **What is the difference between `list.sort` and `list.sort()` ?**
  The first would just return the function itself (i.e. the memory address of the function). The second would actually perform the sort.

Listing 2: Difference between list.sort and list.sort()

```
>>> listA = ['a', 'c', 'b']
>>>
>>> # return the memory address of the function
... # may differ with yours.
... listA.sort
<built-in method sort of list object at 0x000000408BFAE248>
>>>
>>> listA.sort()
>>> ['a', 'b', 'c']
```

- **What is the difference between the operators `is` and `==` ?**
  The `is` operator will determine if two names refer to the same object. This is called "aliasing" and can cause confusing bugs. You can tell if something is the alias of another by using the function `id()`. If the memory addresses of the two names is identical, then one `is` the other. The `==` operator will only determine if the *values* of the two variables are equal. One variable can be `==` to another, but usually one isn't the alias of the other.

- **What does it mean that functions are "first class objects" in Python ?**
  It basically means that functions can be passed as parameters and returned from other functions, just like any ordinary variable.

- **What is "higher order programming" (HOP) ?**
  HOP is the ability to take a function as a parameter into another function, and apply that function parameter against other types of objects. This is particularly useful to process a function against a list. Python has a general purpose HOP command called `map`. Map takes a function and applies it to the other parameters that are passed in.

- **What are dictionaries?**
  Dictionaries are a collection of keys with an associated value attached to them. The keys, as well as the values, can be any kind of Python object. For example ints, string, lists and tuples and other dictionaries, etc. To create a dictionary you separate the keys from the values with a colon (":") and enclose them in curly braces ("{" and "}"). For example `animals = {'a': 'aardvark', 'b': 'baboon', 'c': 'coati'}`.

- **How to I access the value of a particular key?**
  You would code `dictionary_name[key_value]`. In the above example, the code `animals[b]` would return `'baboon'`.

- **Can I get a list of all the keys or all the values in a dictionary?**
  Yes. You would use the method `dictionary_name.keys()` or `dictionary_name.values()`. It doesn't exactly return a list, but something very much like one.

- **What does "Incorrect: Something went wrong: tests don't match up." mean?**
  Occasionally a problem comes up with the grader. Just press the Submit button again until you don't get that message.

- **What is memoization?**
  Memoization is the process of taking a calulation from a function, and saving that value so that if the function is called with the same parameters you don't have to do the calculation all over again. You can just look up in the dictionary to find what the value was that you already computed, thus saving time needed to do the calculation.

- **What are global variables?**
  Normally the variables in a function's "stack frame" are re-initialized each time the function is called. If you need to avoid that, you need to put `global` in front of the variable in both the calling function and the called function. Then the value is carried over from one call to the next.