Chapter 3 - Lists, Tuples and Dictionaries

Python has several other important data types that you'll probably use every day. They are called lists, tuples and dictionaries. This chapter's aim is to get you acquainted with each of these data types. They are not particularly complicated, so I expect that you will find learning how to use them very straight forward. Once you have mastered these three data types plus the string data type from the previous chapter, you will be quite a ways along in your education of Python. You'll be using these four building blocks in 99% of all the applications you will write.

Lists

A Python list is similar to an array in other languages. In Python, an empty list can be created in the following ways.

```
>>> my_list = []
>>> my_list = list()
```

As you can see, you can create the list using square brackets or by using the Python built-in, **list**. A list contains a list of elements, such as strings, integers, objects or a mixture of types. Let's take a look at some examples:

```
>>> my_list = [1, 2, 3]
>>> my_list2 = ["a", "b", "c"]
>>> my_list3 = ["a", 1, "Python", 5]
```

The first list has 3 integers, the second has 3 strings and the third has a mixture. You can also create lists of lists like this:

```
>>> my_nested_list = [my_list, my_list2]
>>> my_nested_list
[[1, 2, 3], ['a', 'b', 'c']]
```

Occasionally, you'll want to combine two lists together. The first way is to use the extend method:

```
>>> combo_list = []
>>> one_list = [4, 5]
>>> combo_list.extend(one_list)
>>> combo_list
[4, 5]
```

A slightly easier way is to just add two lists together.

```
>>> my_list = [1, 2, 3]
>>> my_list2 = ["a", "b", "c"]
>>> combo_list = my_list + my_list2
>>> combo_list
[1, 2, 3, 'a', 'b', 'c']
```

Yes, it really is that easy. You can also sort a list. Let's spend a moment to see how to do that:

```
>>> alpha_list = [34, 23, 67, 100, 88, 2]
>>> alpha_list.sort()
>>> alpha_list
[2, 23, 34, 67, 88, 100]
```

Now there is a got-cha above. Can you see it? Let's do one more example to make it obvious:

```
>>> alpha_list = [34, 23, 67, 100, 88, 2]
>>> sorted_list = alpha_list.sort()
>>> sorted_list
>>> print(sorted_list)
None
```

In this example, we try to assign the sorted list to a variable. However, when you call the **sort()** method on a list, it sorts the list in-place. So if you try to assign the result to another variable, then you'll find out that you'll get a None object, which is like a Null in other languages. Thus when you want to sort something, just remember that you sort them in-place and you cannot assign it to a different variable.

You can slice a list just like you do with a string:

```
>>> alpha_list[0:3]
[2, 23, 34]
```

This code returns a list of just the first 3 elements.

Tuples

A tuple is similar to a list, but you create them with parentheses instead of square brackets. You can also use the **tuple** built-in. The main difference is that a tuple is immutable while the list is mutable. Let's take a look at a few examples:

```
>>> my_tuple = (1, 2, 3, 4, 5)
>>> my_tuple[0:3]
(1, 2, 3)
>>> another_tuple = tuple()
>>> abc = tuple([1, 2, 3])
```

The code above demonstrates one way to create a tuple with five elements. It also shows that you can do tuple slicing. However, you cannot sort a tuple! The last two examples shows how to create tuples using the **tuple** keyword. The first one just creates an empty tuple whereas the second example has three elements inside it. Notice that it has a list inside it. This is an example of **casting**. We can change or **cast** an item from one data type to another. In this case, we cast a list into a tuple. If you want to turn the **abc** tuple back into a list, you can do the following:

```
>>> abc_list = list(abc)
```

To reiterate, the code above casts the tuple (abc) into a list using the **list** function.

Dictionaries

A Python dictionary is basically a **hash table** or a hash mapping. In some languages, they might be referred to as **associative memories** or **associative arrays**. They are indexed with keys, which can be any immutable type. For example, a string or number can be a key. You need to be aware that a dictionary is an unordered set of key:value pairs and the keys must be unique. You can get a list of keys by calling a dictionary instance's **keys** method. To check if a dictionary has a key, you can use Python's **in** keyword. In some of the older versions of Python (2.3 and older to be specific), you will see the **has_key** keyword used for testing if a key is in a dictionary. This keyword is deprecated in Python 2.x and removed entirely from Python 3.x.

Let's take a moment to see how we create a dictionary.

```
>>> my_dict = {}
>>> another_dict = dict()
>>> my_other_dict = {"one":1, "two":2, "three":3}
>>> my_other_dict
{'three': 3, 'two': 2, 'one': 1}
```

The first two examples show how to create an empty dictionary. All dictionaries are enclosed

with curly braces. The last line is printed out so you can see how unordered a dictionary is. Now it's time to find out how to access a value in a dictionary.

```
>>> my_other_dict["one"]
1
>>> my_dict = {"name":"Mike", "address":"123 Happy Way"}
>>> my_dict["name"]
'Mike'
```

In the first example, we use the dictionary from the previous example and pull out the value associated with the key called "one". The second example shows how to acquire the value for the "name" key. Now let's see how to tell if a key is in a dictionary or not:

```
>>> "name" in my_dict
True
>>> "state" in my_dict
False
```

So, if the key is in the dictionary, Python returns a **Boolean True**. Otherwise it returns a Boolean **False**. If you need to get a listing of all the keys in a dictionary, then you do this:

```
>>> my_dict.keys()
dict_keys(['name', 'address'])
```

In Python 2, the **keys** method returns a list. But in Python 3, it returns a *view object*. This gives the developer the ability to update the dictionary and the view will automatically update too. Also note that when using the **in** keyword for dictionary membership testing, it is better to do it against the dictionary instead of the list returned from the **keys** method. See below:

```
>>> "name" in my_dict         # this is good
>>> "name" in my_dict.keys()  # this works too, but is slower
```

While this probably won't matter much to you right now, in a real job situation, seconds matter. When you have thousands of files to process, these little tricks can save you a lot of time in the long run!

Wrapping Up

In this chapter you just learned how to construct a Python list, tuple and dictionary. Make sure you understand everything in this section before moving on. These concepts will assist you in designing your programs. You will be building complex data structures using these

building blocks every day if you choose to pursue employment as a Python programmer. Each of these data types can be nested inside the others. For example, you can have a nested dictionary, a dictionary of tuples, a tuple made up of several dictionaries, and on and on.

When you are ready to move on, we will learn about Python's support for conditional statements.