1-3_Lists Sets and Dictionaries

September 15, 2015

1 Compound Data Types: Lists, Dictionaries, Sets, Tuples

Based on materials by Milad Fatenejad, Joshua R. Smith, and Will Trimble

The ability to refer to *collections* of data will be critical to almost any science - at least if you're collecting more than a single measurement of something.

The two main collections are **lists** and **dictionaries**, but I'll mention **sets** and **tuples** as well. I'll also go over reading text data from files.

1.1 Lists

A list is an ordered, indexable collection of data. Lets say you're doing a study on the following countries:

```
country:

"Afghanistan"
"Canada"
"Sierra Leone"
"Denmark"
"Japan"
```

So you could put that data into lists like

```
In [1]: country_list = ["Afghanistan", "Canada", "Sierra Leone", "Denmark", "Japan"]
```

obviously `country_list` is of type list:

```
In [2]: type(country_list)

Out[2]: list
```

Sometimes you want to know how many items are in a list. Use the `len` command.

```
In [3]: len(country_list)

Out[3]: 5
```

1.2 Indexing

Lists can be thought of as a series of boxes. Each box has a different value.

Python lists have the charming (annoying?) feature that they are indexed from zero. Therefore, to find the value of the first item in `country_list`:

```
In [4]: country_list[0]

Out[4]: 'Afghanistan'
```

And to find the value of the third item

```
In [5]: country_list[2]
```

```
Out[5]: 'Sierra Leone'
```

Lists can be indexed from the back using a negative index. The last item of currentList

```
In [6]: country_list[-1]
```

```
Out[6]: 'Japan'
```

and the next-to-last

```
In [7]: country_list[-2]
```

```
Out[7]: 'Denmark'
```

You can "slice" items from within a list. Lets say we wanted the second through fourth items from voltageList

```
In [8]: country_list[1:4]
```

```
Out[8]: ['Canada', 'Sierra Leone', 'Denmark']
```

Or from the third item to the end

```
In [9]: country_list[2:]
```

```
Out[9]: ['Sierra Leone', 'Denmark', 'Japan']
```

and so on.

1.2.1 List Methods

Just like strings have methods, lists do too. IPython lets us do tab completion after a dot ('.') to see what an object has to offer. Try it now!

```
In [10]: list.
```

```
          File "<ipython-input-10-a7c0b3cf0b01>", line 1
        list.
             ^
    SyntaxError: invalid syntax
```

One useful method is append. Lets say we want to stick the following countries on the end of our list:

```
country:

"United States"
"Brazil
```

If you want to append items to the end of a list, use the append method.

```
In [11]: country_list.append("United States")
```

```
In [12]: country_list.append("Brazil")

In [13]: country_list

Out[13]: ['Afghanistan',
          'Canada',
          'Sierra Leone',
          'Denmark',
          'Japan',
          'United States',
          'Brazil']
```

You can see how that approach might be tedious in certain cases. If you want to concatenate a list onto the end of another one, use extend.

```
In [14]: country_list.extend(["Vietnam","South Africa"])

In [15]: country_list

Out[15]: ['Afghanistan',
          'Canada',
          'Sierra Leone',
          'Denmark',
          'Japan',
          'United States',
          'Brazil',
          'Vietnam',
          'South Africa']
```

Lists have many more methods. Try to see what the following methods do:

```
In [16]: # Uncomment and try me out!
         # country_list.sort()
         # country_list.index()
         # country_list.remove()
```

1.2.2 Heterogeneous Data

Lists can contain hetergeneous data.

```
In [17]: demolist = ['life', \
                     42, \
                     'the universe', \
                     [1,2,3], \
                     {"year": 2015}]
```

We've got strings, ints, dictionaries, and even other lists in there. The slashes are there so we can continue on the next line. They aren't necessary but they can sometimes make things look better.

```
In [18]: print demolist

['life', 42, 'the universe', [1, 2, 3], {'year': 2015}]
```

Notice that object in the list has its own index:

```
In [19]: demolist[3]

Out[19]: [1, 2, 3]
```

But we can replace any item in a list using its index:

```
In [20]: demolist[3] = "Nevermind"
         demolist

Out[20]: ['life', 42, 'the universe', 'Nevermind', {'year': 2015}]
```

1.2.3 Explanation:

The above resolves to *False* because when you call `a + b` it generates a new object. So even though the contents are the same, it is referring to different objects.

There's a ton more to know about lists, but lets press on.

1.3 Tuples

Tuples are another of python's basic compound data types that are almost like lists. The difference is that a tuple is immutable; once you set the data in it, the tuple cannot be changed. You define a tuple as follows.

```
In [21]: tup = ("red", "white", "blue")

In [22]: type(tup)

Out[22]: tuple
```

You can slice and index the tuple exactly like you would a list. Tuples are used in the inner workings of python, and a tuple can be used as a key in a dictionary, whereas a list cannot as we will see in a moment.

See if you can retrieve the third element of **tup**:

```
In [ ]:

In [23]: ## What happens when you try to do the following? (Remember a tuple is immutable)

         tup[0] = 'aquamarine'


        ---------------------------------------------------------------------------

        TypeError                                 Traceback (most recent call last)

        <ipython-input-23-0bb393cdc96c> in <module>()
          1 ## What happens when you try to do the following? (Remember a tuple is immutable)
          2
    ----> 3 tup[0] = 'aquamarine'


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

1.4 Sets

Most introductory python courses (including Codecademy) do not go over sets this early (or at all), but I've found this data type to be useful. The python set type is similar to the idea of a mathematical set: it is an unordered collection of unique things. Consider:

```
In [24]: mystuff = [1,2,3,4,4,4,4,4,4,4,4] # I like fours, even if this is a little redundant
         my_unique_stuff = set(mystuff)

In [25]: ## how many items are in my_unique_stuff?

In [26]: # Note: the {} notation for sets is new in python 2.7.x
         # For older pythons, you must use set(['apple', 'banana', ...])
         fruit = {"apple", "banana", "pear", "banana"}
```

Since sets contain only unique items, there's only one banana in the set fruit.

```
In [27]: print fruit
         ## what do you think happens when you try to get the first item in this set?
         ## fruit[0]
```

```
set(['pear', 'banana', 'apple'])
```

You can do things like intersections, unions, etc. on sets just like in math. Here's an example of an intersection of two sets (the common items in both sets).

```
In [28]: firstBowl = {"apple", "banana", "pear", "peach"}
```

```
In [29]: secondBowl = {"peach", "watermelon", "orange", "apple"}
```

Set operations can be performed with functions from the set class:

```
In [30]: set.intersection(firstBowl, secondBowl)
```

```
Out[30]: {'apple', 'peach'}
```

Or, you can use methods on one of your sets:

```
In [31]: firstBowl.intersection(secondBowl)
```

```
Out[31]: {'apple', 'peach'}
```

You can check out more info using the help docs.

1.5 Dictionaries

A python dictionary is a collection of key, value pairs. The key is a way to name the data, and the value is the data itself. Dictionaries are very powerful, especially when working with data

```
In [32]: poets_dict = {"name": "Forough Farrokhzad", \
                       "year of birth": 1935, \
                       "year of death": 1967, \
                       "place of birth": "Iran", \
                       "language": "Persian", \
                       "works": ["Remembrance of a Day","Unison","The Shower of Your Hair","Portrait of F
```

```
In [33]: print poets_dict
```

```
{'name': 'Forough Farrokhzad', 'language': 'Persian', 'year of birth': 1935, 'place of birth': 'Iran',
```

```
In [34]: # a dictionary has keys and values. You use the keys to access the values
         print poets_dict.keys()
         print poets_dict.values()
```

```
['name', 'language', 'year of birth', 'place of birth', 'works', 'year of death']
['Forough Farrokhzad', 'Persian', 1935, 'Iran', ['Remembrance of a Day', 'Unison', 'The Shower of Your
```

If you wanted the value of a particular key:

```
In [35]: poets_dict["name"]
```

```
Out[35]: 'Forough Farrokhzad'
```

Or perhaps you wanted the last element of the `works` list

```
In [36]: poets_dict["works"][-1]
```

```
Out[36]: 'Portrait of Forough'
```

Once a dictionary has been created, you can change the values of the data if you like.

```
In [37]: poets_dict["language"] = "Farsi"
```

You can also add new keys to the dictionary. Note that dictionaries are indexed with square braces, just like lists–they look the same, even though they're very different.

```
In [38]: poets_dict["gender"] = "Female"
```

Dictionaries, like strings, lists, and all the rest, have built-in methods. (We saw this above when we accessed the **keys** and **values**.

You can also get a list of keys *and* values

```
In [39]: poets_dict.items()
```

```
Out[39]: [('name', 'Forough Farrokhzad'),
          ('language', 'Farsi'),
          ('gender', 'Female'),
          ('year of birth', 1935),
          ('place of birth', 'Iran'),
          ('works',
           ['Remembrance of a Day',
            'Unison',
            'The Shower of Your Hair',
            'Portrait of Forough']),
          ('year of death', 1967)]
```

One thing to be careful of, is an **inplace** method dict.update().

This also allows you to add something to your dictionary. It does **NOT** return a copy of the dictionary with the new key:value added. Instead it just updated your current variable.

```
In [40]: ## so look at this example
         new_data_dict = poets_dict.update({'newthing':42})
         ## What does the variable **new_data_dict** contain??  Why?

         ## What does data_dict contain??


         # What would happen if you did this?
         # data_dict = data_dict.update({'newkey':'newvalue'})
```

1.6 Checking if something is in a container

To see if something is in a container, use the `in` operator. This works for all the types of container we described above:

```
In [41]: print "Afghanistan" in country_list # lists
         print "blue" in tup       # tuples
         print "mango" in fruit    # sets

         # For dictionaries, this only works for *keys*
         # Can you work out how to check for a value?
         print "language" in poets_dict
```

```
True
True
False
True
```