05-Lists Sets and Dictionaries

Unknown Author

August 20, 2013

1 Compound Data Types: Lists, Dictionaries, Sets, Tuples, and Reading Files

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

Python would be a fairly useless language if it weren't for the compound data types. The main two 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 have collected some current and voltage data that looks like this:

```
voltage:

-2.0

-1.0

0.0

1.0

2.0


current:

-1.0

-0.5

0.0

0.5

1.0
```

So you could put that data into lists like

```
In [1]: voltageList = [-2.0, -1.0, 0.0, 1.0, 2.0]

        currentList = [-1.0, -0.5, 0.0, 0.5, 1.0]
```

obviously voltageList is of type list:

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

Out [2]:
        list
```

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

```
In [3]: voltageList[0]

Out [3]:
        -2.0
```

And to find the value of the third item

```
In [4]: voltageList[2]

Out [4]:
        0.0
```

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

```
In [5]: currentList[-1]

Out [5]:
        1.0
```

and the next-to-last

```
In [6]: currentList[-2]

Out [6]:
        0.5
```

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

```
In [7]: voltageList[1:4]

Out [7]:
        [-1.0, 0.0, 1.0]
```

Or from the third item to the end

```
In [8]: voltageList[2:]
```

Out [8]:
```
[0.0, 1.0, 2.0]
```

and so on.

Append and Extend

Just like strings have methods, lists do too.

In [9]:
```
dir(list)
```

Out [9]:
```
['__add__',
 '__class__',
 '__contains__',
 '__delattr__',
 '__delitem__',
 '__delslice__',
 '__doc__',
 '__eq__',
 '__format__',
 '__ge__',
 '__getattribute__',
 '__getitem__',
 '__getslice__',
 '__gt__',
 '__hash__',
 '__iadd__',
 '__imul__',
 '__init__',
 '__iter__',
 '__le__',
 '__len__',
 '__lt__',
 '__mul__',
 '__ne__',
 '__new__',
 '__reduce__',
 '__reduce_ex__',
 '__repr__',
 '__reversed__',
 '__rmul__',
 '__setattr__',
 '__setitem__',
 '__setslice__',
 '__sizeof__',
 '__str__',
 '__subclasshook__',
 'append',
 'count',
 'extend',
 'index',
 'insert',
```

```
 'pop',
 'remove',
 'reverse',
 'sort']
```

One useful method is append. Lets say we want to stick the following data on the end of both our lists.

voltage:

3.0

4.0

current:

1.5

2.0

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

```
In [10]: voltageList.append(3.)
```

```
In [11]: voltageList.append(4.)
```

```
In [12]: voltageList
```

```
Out [12]:
         [-2.0, -1.0, 0.0, 1.0, 2.0, 3.0, 4.0]
```

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 [13]: currentList.extend([1.5, 2.0])
```

```
In [14]: currentList
```

```
Out [14]:
         [-1.0, -0.5, 0.0, 0.5, 1.0, 1.5, 2.0]
```

Length of Lists

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

```
In [15]: len(voltageList)
```

```
Out [15]:
         7
```

Heterogeneous Data

Lists can contain hetergeneous data.

In [16]:
```
dataList = ["experiment: current vs. voltage", \
    "run", 47, \
    "temperature", 372.756, \
    "current", [-1.0, -0.5, 0.0, 0.5, 1.0], \
    "voltage", [-2.0, -1.0, 0.0, 1.0, 2.0]]
```

In [17]:
```
print dataList
```

```
['experiment: current vs. voltage', 'run', 47, 'temperature', 372.756,
'current', [-1.0, -0.5, 0.0, 0.5, 1.0], 'voltage', [-2.0, -1.0, 0.0,
1.0, 2.0]]
```

We've got strings, ints, floats, 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.

1.2 Assigning Variables to Other Variables

Something that might cause you headaches in the future is how python deals with assignment of one variable to another. When you set a variable equal to another, both variables point to the same thing. Changing the first one ends up changing the second. Be careful about this fact.

In [18]:
```
a = [1,2]
```

In [19]:
```
b = a
```

In [20]:
```
a.append(10)
```

In [21]:
```
b
```

Out [21]:
```
[1, 2, 10]
```

There's a ton more to know about lists, but lets press on. Check out Dive Into Python or the help documentation for more info.

1.3 Reading From Files

At this point it is useful to take a detour regarding files. Lets say you have a file with some current and voltage data and some metadata.

data.dat:

```
experiment: current vs. voltage

run: 47

temperature: 372.756

current: [-1.0, -0.5, 0.0, 0.5, 1.0]

voltage: [-2.0, -1.0, 0.0, 1.0, 2.0]
```

We can read this data into a list type variable pretty easily.

In [22]:
```
f = open("data.dat")
```

```
        ---------------------------------------------------------------
-------------
    IOError                                   Traceback (most recent
call last)

        <ipython-input-22-68417d065ab1> in <module>()
    ----> 1 f = open("data.dat")

        IOError: [Errno 2] No such file or directory: 'data.dat'
```

In []:
```
ivdata = f.readlines()
```

In [23]:
```
f.close()
```

```
        ---------------------------------------------------------------
-------------
    NameError                                 Traceback (most recent
call last)

        <ipython-input-23-fa01cc0d8510> in <module>()
    ----> 1 f.close()
```

```
        NameError: name 'f' is not defined
```

In [24]:
```
ivdata
```

```
        ---------------------------------------------------------------
-------------
    NameError                                 Traceback (most recent
call last)



        <ipython-input-24-706be465a63f> in <module>()
    ----> 1 ivdata



        NameError: name 'ivdata' is not defined
```

Right now the data in ivdata isn't in a particularly useful format, but you can imagine that with some additional programming we could straighten it out. We will eventually do that.

1.4 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 [25]:
```
tup = ("red", "white", "blue")
```

In [26]:
```
type(tup)
```

Out [26]:
```
        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 [26]:
```

```

1.5 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 [27]:
```
# 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"}
```

ou have to use a list to create a set.

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

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. We won't be returning to sets, but its good for you to know they exist.

1.6 Dictionaries

Recall our file data.dat which contained our current-voltage data and also some metadata. We were able to import the data as a list, but clearly the list type is not the optimal choice for a data model. The dictionary is a much better choice. 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. Here's a way to create a dictionary that contains all the data in our data.dat file in a more sensible way than a list.

In [32]:
```
dataDict = {"experiment": "current vs. voltage", \
                    "run": 47, \
                    "temperature": 372.756, \
                    "current": [-1.0, -0.5, 0.0, 0.5, 1.0], \
                    "voltage": [-2.0, -1.0, 0.0, 1.0, 2.0]}
```

In [33]:
```
print dataDict
```

```
{'current': [-1.0, -0.5, 0.0, 0.5, 1.0], 'experiment': 'current vs.
voltage', 'run': 47, 'temperature': 372.756, 'voltage': [-2.0, -1.0,
0.0, 1.0, 2.0]}
```

This model is clearly better because you no longer have to remember that the run number is in the second position of the list, you just refer directly to "run":

```
In [34]: dataDict["run"]
```

```
Out [34]:
47
```

If you wanted the voltage data list:

```
In [35]: dataDict["voltage"]
```

```
Out [35]:
[-2.0, -1.0, 0.0, 1.0, 2.0]
```

Or perhaps you wanted the last element of the current data list

```
In [36]: dataDict["current"][-1]
```

```
Out [36]:
1.0
```

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

```
In [37]: dataDict["temperature"] = 3275.39
```

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]: dataDict["user"] = "Johann G. von Ulm"
```

Dictionaries, like strings, lists, and all the rest, have built-in methods. Lets say you wanted all the keys from a particular dictionary.

```
In [39]: dataDict.keys()
```

```
Out [39]:
['run', 'temperature', 'current', 'experiment', 'user', 'voltage']
```

also, values

```
In [40]: dataDict.values()
```

```
Out [40]:
[47,
 3275.39,
 [-1.0, -0.5, 0.0, 0.5, 1.0],
 'current vs. voltage',
```

```
 'Johann G. von Ulm',
 [-2.0, -1.0, 0.0, 1.0, 2.0]]
```

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

In [41]:
```
dataDict.items()
```

Out [41]:
```
[('run', 47),
 ('temperature', 3275.39),
 ('current', [-1.0, -0.5, 0.0, 0.5, 1.0]),
 ('experiment', 'current vs. voltage'),
 ('user', 'Johann G. von Ulm'),
 ('voltage', [-2.0, -1.0, 0.0, 1.0, 2.0])]
```