Data Structures

PS 452: Text as Data

Fall 2014

Department of Political Science, Stanford University

Created by **Frances Zlotnick**

Please contact me at zlotnick@stanford.edu with questions or comments.

This tutorial provides a brief introduction to the basic python collection data structures, including:

- lists
- tuples
- dicts
- sets

Many tasks can be accomplished in several different ways, but selecting appropriate data structures can make a huge difference in the efficiency and speed of your code, and cut down on unnecessary work. You should choose the structure that is optimized for the particular task you need it to do. Some things to think about:

- What is the primary purpose of this collection? Does it simply need to store items, or does it need to be searchable?
- Will you need to modify the collection as you work with it, or will it remain static?
- Do you need to preserve the ordering of the elements?
- How big is your data? Many of the differences between these structures are trivial with small ammounts of data, but become important as your data grows.

Some general tips:

1. Searching is always more efficient with hashed objects. Searching a hashed object will always be faster than looping over an unhashed `list`.
2. Take advantage of python's native hierarchical data structures. `Dicts` are extraordinarily efficient and useful (when appropriate) and it is well worth the effort to learn how to use them.
3. Don't underestimate the difference that efficient code can make. Inefficient code on big data will eat up all of your RAM and cause your code to crash.

Lists

The workhorse data structure in python is the `list`. `Lists` are instantiated using square brackets, with elements separated by commas.

In [1]:
```
mylist = ["This", "is", "a", "list", "of", "strings."]
```

Unlike `vectors` in `R`, python `lists` can contain heterogenous data types.

In [2]:
```
myHeterogenousList = ["string", 15, "another string", 2.0]
```

`Lists` are nestable, and a list can contain any type of collection as an element, and you can easily iterate over list elements.

In [3]:
```
#we'll discuss these other data structures a bit later
nestedList = ["apple", 2.0, [1,2,3], {'dictionaryKey': 'dictionaryValue'}, set(['setElement']) ]
```

In [4]:
```
for element in nestedList:
    print type(element)
```

```
<type 'str'>
<type 'float'>
<type 'list'>
<type 'dict'>
<type 'set'>
```

You may come across example of list comprehension, which is a handy syntax for creating a list from another list. It is basically a one line `for` loop. For our purposes, there is no reason to use this syntax instead of a for loop unless you care about saving a little bit of typing. If you find it confusing, just use a for loop. There are analagous methods for `sets` and `dicts` as well.

In [5]:
```
veggies = ["carrot", "potato","spinach", "turnip"]
VEGGIES = [x.upper() for x in veggies]
print VEGGIES
```

```
['CARROT', 'POTATO', 'SPINACH', 'TURNIP']
```

`Lists` are ordered, which means they are indexible. You can locate a particular object using the `index` method.

In [6]:
```
print veggies[1:3]
veggies.index("spinach")
```

```
['potato', 'spinach']
```

Out[6]: 2

`Lists` are mutable, which means they can be modified and those changes will persist without being explicitly saved. You can add to the end of a list using `append`, or at a specific index using `insert

In [7]:
```
veggies.append("zucchini")
print veggies
```

```
['carrot', 'potato', 'spinach', 'turnip', 'zucchini']
```

In [8]:
```
veggies.insert(3, "kale")
print veggies
```

```
['carrot', 'potato', 'spinach', 'kale', 'turnip', 'zucchini']
```

You can `reverse` a `list` or `sort` it.

In [9]:
```
veggies.reverse()
print veggies
```

```
['zucchini', 'turnip', 'kale', 'spinach', 'potato', 'carrot']
```

In [10]:
```
veggies.sort()
print veggies
```

```
['carrot', 'kale', 'potato', 'spinach', 'turnip', 'zucchini']
```

`remove` will delete a particular item from a list.

In [11]:
```
veggies.remove("turnip")
print veggies
```

```
['carrot', 'kale', 'potato', 'spinach', 'zucchini']
```

`pop` will remove the last item from the list and return it.

In [12]:
```
a = veggies.pop()
print a
print veggies
```

```
zucchini
['carrot', 'kale', 'potato', 'spinach']
```

You can count the number of times a particular object appears in a list with `count`.

In [13]:
```
veggies.append("potato")
veggies.count("potato")
```

Out[13]: 2

You can find the smallest element with `min`, the largest with `max`, and the total number of items with `len`. For string variables, the min and max correspond to their alphabetical order.

In [14]:
```
print min(veggies)
print max(veggies)
print len(veggies)
```

```
carrot
spinach
5
```

Tuples

`Tuples` are variant on `lists`; the main difference is that they are immutable. If you have no need to change a `list`, or if you'd like to to use a `list` but can't because your purpose requires an immutable object (for instance, as a dictionary key -- more on this below), use a `tuple`.

In [15]:
```
myTuple = (1, 2, 3, 4, 5)
myTuple.append(6)
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-15-8f1c3896710e> in <module>()
      1 myTuple = (1, 2, 3, 4, 5)
----> 2 myTuple.append(6)

AttributeError: 'tuple' object has no attribute 'append'
```

You can, however, search, index, and find attributes of `tuples` just as you would `lists`.

In [16]:
```
myTuple[1:3]
```

Out[16]: (2, 3)

In [17]:
```
max(myTuple)
```

Out[17]: 5

In [18]:
```
myTuple.index(3)
```

Out[18]: 2

Dicts

Dictionaries or `dicts` are hierarchical data structures that map items, consisting of keys and their corresponding values. Unlike `lists`, `dicts` are hashed, which means they are much faster and more efficient to search through. A `dict` is created with curly braces. There are a few ways of adding to a dictionary:

In [19]:
```
myDict = {"peanut butter": "jelly"}

#is equivalent to
myDict = {}
myDict["peanut butter"] = "jelly"
```

While values can be any data type or structure, keys may only be immutable objects.

In [20]:
```
nums = {[1,2] : "one and two"}
nums
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-20-f85625347d97> in <module>()
----> 1 nums = {[1,2] : "one and two"}
      2 nums

TypeError: unhashable type: 'list'
```

The principle function of `dicts` is to facilitate quick searching of associated values. If, say, I had a `dict` of phone numbers like the following:

In [21]:
```
phoneNumbers = { 'joe' : 123456, 'mary': 5436789, 'phil': 2463579}
```

I can find Joe's number by looking up the value associated with his name. As your data gets larger, this becomes much, much faster than looping over a list and checking whether each item is what you are looking for.

In [22]:
```
phoneNumbers['joe']

#is equivalent to
phoneNumbers.get("joe")
```

Out[22]: 123456

You can see all of the key value pairs in your `dict` with `items`.

In [23]:
```
phoneNumbers.items()
```

Out[23]: [('phil', 2463579), ('joe', 123456), ('mary', 5436789)]

`keys` and `values` return all of the keys contained in your dictionary and all of the values, respectively

In [24]:
```
phoneNumbers.keys()
```

Out[24]: ['phil', 'joe', 'mary']

In [25]:
```
phoneNumbers.values()
```

Out[25]: [2463579, 123456, 5436789]

You can check whether a key already exists in your dictionary with `has_key`. Note that keys must be unique, but values need not be.

In [26]:
```
phoneNumbers.has_key("louise")
```

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

`Dicts` can be nested, which means that they can contain other `dicts` as values.

In [27]:
```
phoneNumbers["bill"] = {}
```

In [28]:
```
phoneNumbers["bill"]["home"] = 1234567
phoneNumbers["bill"]["cell"] = 9876543
phoneNumbers
```

Out[28]:
```
{'bill': {'cell': 9876543, 'home': 1234567},
 'joe': 123456,
 'mary': 5436789,
 'phil': 2463579}
```

`Dicts` are unordered, which means they cannot be indexed. It does not make sense to ask for the "first" or "last" item in a dictionary because there is no inherent ordering to them.

In [29]:
```
phoneNumbers[2]
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-29-c9841e947201> in <module>()
----> 1 phoneNumbers[2]

KeyError: 2
```

You can iterate over dictionaries using `iteritems`, but since they are unordered, the order they will follow is not defined, and not reliable.

In [30]:
```
for a in phoneNumbers.iteritems():
    print a
```

```
('phil', 2463579)
('joe', 123456)
('bill', {'cell': 9876543, 'home': 1234567})
('mary', 5436789)
```

Sets

Sets are unordered, hashed collections with no duplicate elements. This makes them extremely efficient for situations when you simply want to test whether a certain object exists in your data or not.

In [31]:
```
mySet = set()
mySet.add("hello")
mySet.add("hi")
mySet.add("hey")
print mySet
```

```
set(['hi', 'hello', 'hey'])
```

In [32]:
```
print "hi" in mySet
print "yo" not in mySet
```

```
True
True
```

Adding to a set an item that already exists in it will not do anything.

In [33]:
```
mySet.add("hello")
print mySet
```

```
set(['hi', 'hello', 'hey'])
```

Keep in mind that trying to `remove` an element that does not exist in a `set` will throw an error, so if you are unsure if an item is in a set, either check first, or use `discard`.

In [34]:
```
mySet.remove("yo")
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-34-37a72fc7381d> in <module>()
----> 1 mySet.remove("yo")

KeyError: 'yo'
```

In [35]:
```
mySet.discard("yo")
```

There are a number of useful `set` operations.

In [36]:
```
set1 = set([1,2,3])
set2 = set([3,4,5])
```

`difference` or `-` will return a `set` of the elements in the first `set` and not in the second `set`.

In [37]:
```
set1 - set2

#is equivalent to
set1.difference(set2)
```

Out[37]: {1, 2}

`union` or `|` will return a `set` of the elements in either `set`.

In [38]:
```
set1 | set2

#is equivalent to
set1.union(set2)
```

Out[38]: {1, 2, 3, 4, 5}

`intersection` or `&` will return a `set` of the elements in both `sets`

In [39]:
```
set1 & set2

#is equivalent to
set1.intersection(set2)
```

Out[39]: {3}

`symmetric_difference` or `^` will return a `set of the elements in one set or the other but not both.

In [40]:
```
set1 ^ set2

#is equivalent to
set1.symmetric_difference(set2)
```

Out[40]: {1, 2, 4, 5}

As with `dicts` you can iterate over a `set`, but the order may be unpredictable because it is an inherently unordered structure.

In [41]:
```
for a in mySet:
    print a
```

```
hi
hello
hey
```

Summary

Lists

- Ordered
- Nestable
- Mutable

Tuples

- Ordered
- Nestable
- Immutable

Dicts

- Unordered
- Nestable
- Mutable
- Key-value pairs
- Keys must be immutable
- Hashed

Sets

- Unordered
- Mutable
- Hashed