Hierarchical and Associative Data Structures in Python vs R

or How I learned to Stop Worrying and Love the Dictionary

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 document supplements the Data Structures tutorial by providing some additional rationale and examples for the use of the `dictionary` structure in python. It is written for `R` users who have a basic understanding of programming but no formal computer science training.

If you are already convinced, you can skip to the applied section on using nested dictionaries. If you want to believe, keep reading.

Why use `dictionaries`?

One of the major differences between `R` and python is that python has no native data frame structure. `R` users often try to approximate data frames in python by using nested lists or multiple unassociated lists. This is a bad idea for a number of reasons:

1. It is likely to introduce errors into your data.
2. It is slow.
3. It fails to take advantage of python's native data structures that are optimized for this task.

Python's analgous data structure is the `dictionary`, which holds pairs of values in a hashed object that is virtually instant to search over. The key to understanding dictionaries is to think of your data as a collection of observations with associated attributes, rather than a 2-dimensional table of values.

You can write a table of data in the following way

Fruit	color	citrus	Good for pie
"Apple"	"red"	0	"yes"
"Orange"	"orange"	1	"no"
"Banana"	"yellow"	0	"maybe"
"Grape"	"green"	0	"no"
"Lemon"	"yellow"	1	"maybe"

You can, equivalently, write exactly the same information the following way.

Apple:
color: "red"
citrus: 0
Good for pie: "yes"

Orange:
color: "orange"
citrus: 1
Good for pie: "no"

and so on. This seems pretty obvious, but it is a significant change in how to think about data, and it takes practice to start thinking thinking of your data as objects with attributes rather than rows and columns of a table. While it might be a philosophical distinction for some purposes, it has important implications for approaching computational tasks.

R dataframes and vectors versus python lists

Data frames in `R` are sets of linked vectors. Under the hood, `dataframes` are are a subset of `R` `lists` with a few restrictions imposed, like that all of the vectors must be of the same length. Operations on `dataframes` maintain the overall associative structure between the constituent vectors.

In [1]:
```
%%capture capt
import rpy2, os
os.chdir("/Users/franceszlotnick/Dropbox/TextAsData/Sections/Tutorials/")
%load_ext rpy2.ipython
```

In [2]:
```
%R vecA<- c("A","B","C")
%R vecB<-  c(83,2,15)
%R vecC<- c("x", "y","z")
%R df<- data.frame(vecA, vecB, vecC)
%R print(df)
```

```
  vecA vecB vecC
1    A   83    x
2    B    2    y
3    C   15    z
```

If you reorder a dataframe, the relative location of all of the observations shift to maintain the appropraite structure.

In [3]:
```
%R print(df[order(vecB),])
```

```
  vecA vecB vecC
2    B    2    y
3    C   15    z
1    A   83    x
```

Python does not natively have a data frame structure. There is no way to link various `lists` to eachother, so they are all standalone objects. `R` users often try to approximate data frames in python by using nested `lists` or just many `lists` with associated observations stored at equivalent indices. This is a bad idea because, unlike `R` `vectors`, **python lists are mutable**.

In `R`, vectors are immutable. Performing operations on vectors reuturns an entirely new vector, it doesn't change the underlying data.

In [4]:
```
%R vec<- c(1:10)
%R print(vec)
%R print(sort(vec, decreasing=T))
%R print(vec)
```

```
 [1]  1  2  3  4  5  6  7  8  9 10

 [1] 10  9  8  7  6  5  4  3  2  1

 [1]  1  2  3  4  5  6  7  8  9 10
```

To modify a `vector`, you have to apply some function and then save the returned object over the existing variable name.

In [5]:
```
%R vec<- sort(vec, decreasing=T)
%R print(vec)
```

```
 [1] 10  9  8  7  6  5  4  3  2  1
```

Python lists are mutable, which means you can sort, add or remove elements, and perform many other functions on a list without explictly saving those changes and it will change the underlying object.

In [6]:
```
exampleList = range(1,11)
exampleList
```

Out[6]:
```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

In [7]:
```
exampleList.sort(reverse=True)
exampleList
```

Out[7]:
```
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

For this reason, it is dangerous to rely solely on observations being located at equivalent indices in separate lists; it extremely easy to destroy the structure of your data. If you accidentally reorder or otherwise change elements in one list, you will not be able to match observations across lists.

Lists vs Dicts

Another reason to use `dictionaries` is that they are much faster to search through than `lists`. `Dicts` are *hashed* objects, which means that their keys are stored at a particular location in memory according to a hashing function that takes the input and maps it to a memory address. The details of this aren't important for our purposes, but the result is that searching for a particular element in a hashed object is virtually instant. Lists are not hashed, so searching over a list requires the computer must begin at index 0 and check every element sequentially.

Computer scientists evaluate the complexity of algorithms and operations based on how long it takes to run as the data approaches size N, and express this through "Big O Notation." Searching over a `list` is O(N), while searching over a `dict` or a `set` is O(1). This means that as your data grows, searching over a `list` will take longer and longer, while the speed of searching over a hashed object will stay constant.

In [8]:
```
x = range(0,100000)
y = set(x)
```

In [9]:
```
%%timeit
#searching over a list
80000 in x
```

```
1000 loops, best of 3: 1.9 ms per loop
```

In [10]:
```
%%timeit
#searching over a set
80000 in y
```

```
10000000 loops, best of 3: 129 ns per loop
```

Note the units on those results. There are 1 million nanoseconds in a milisecond. Searching through a list takes over *10,000* times longer than searching through an equivalent hashed structure. This is a trivial example, but since you perform many hundreds of operations on objects of significant length in a given script, this time adds up and can slow down your code to point of hanging.

Let's look at another example that represents a task we often encounter. Say we have have a set of cases with one associated value for each case. I want to check whether some other set of cases exist in my data, and if so, get the associated value, and put the pair into another list as a tuple.

We can represent this as two lists, or as a dictionary. Let's examine the list method first.

In [11]:
```
import random


#create list of cases
caseid = []
for i in range(0, 25000):
    caseid.append("FJZ" + str(i))

#create list of paired observations. We'll make up some random data.
observations = []
for i in range(0, 25000):
    observations.append(random.randint(1,10))

#create a list of cases to search for in our data. Some of these won't actually be in our data.
casesToFind = []
for i in range(100):
    casesToFind.append("FJZ" + str(random.randint(0,50000)))
```

Now we'll loop over our list of caseid's, checking whether each one corresponds to something in our data. If so, we'll find what index it exists at, find the observation in the corresponding list at that same index, and add the pair to some other list as a tuple.

In [12]:
```
%%timeit
found = []
for i in casesToFind:
    if i in caseid:
        whereAt = caseid.index(i)
        found.append((i, observations[whereAt]))
```

```
10 loops, best of 3: 50.4 ms per loop
```

Now let's do the equivalent task with a dicitonary.

In [13]:
```
#put our data into a dictionary with the caseids as keys and observations as values
d = {}

for i in range(0, len(caseid)):
    d[caseid[i]] = observations[i]
```

We once again loop over our search list, but now we check whether the element exists as a key in our dicitonary, and if so, append the key and associated value to our found `list`.

In [14]:
```
%%timeit
found = []
for i in casesToFind:
    if d.has_key(i):
        found.append((i, d[i]))
```

```
10000 loops, best of 3: 30 µs per loop
```

This task is more than *1500* times faster with a dicitonary than with lists.

Now that you know all about why you *should* use dictionaries, let's talk a little more about how to use nested dictionaries. The logic is the same as a regular dicitonary, but it can be tricky to get the hang of it.

Nested Dictionaries

Because `dicts` can hold any type of object or collection as values, it is possible to store other `dicts` as values. This type of nesting is very useful if you want to be able to iterate over objects that have many associated values.

Lets begin with a small example: I have a set of information about various countries around the world. Since the data is associated, it makes the most sense to store these data in dictionaries.

In [15]:
```
usa = {"population": 318968000,
       "majority language": "english",
       "GDP per cap": "$53,001",
       "capital": "Washington, D.C."
       }

poland = {"population": 38485779,
       "majority language": "polish",
       "GDP per cap": "$21,118",
       "capital": "Warsaw"
       }

uzbekistan = {
        "population": 30185000,
       "majority language": "uzbek",
       "GDP per cap": "$4,038",
       "capital": "Tashkent"
        }

tanzania = {
        "population": 47400000,
       "majority language": "swahili",
       "GDP per cap": "$1,813",
       "capital": "Dar es Salaam"
        }
```

Now if want a list of the capitals of all of the countries, I can easily pull that out.

In [16]:
```
print usa["capital"]
print poland["capital"]
print uzbekistan["capital"]
print tanzania["capital"]
```

```
Washington, D.C.
Warsaw
Tashkent
Dar es Salaam
```

It would be nice if I didn't have to do all that typing to do exactly the same task, but since these are unlinked objects there is no simple syntax for doing that. If we put them all into the same structure, however, we can do this very simply. Since these are all similar objects (i.e. dicitonaries with country data), it makes sense for us to nest all of these inside of a larger "countries" dictionary that holds each of these country-specific dictionaries as a value.

More conceptually, though, what we have with the separate dictionaries are the equivalent of *unlinked rows on a data set*. We want to bind these "rows" together to form a complete dataset.

In [17]:
```
countries = {}
countries["usa"] = usa
countries["poland"] = poland
countries["uzbekistan"] = uzbekistan
countries["tanzania"] = tanzania
```

Now we can iterate over our outer dictionary and print out all of the capitals without all of the repetitive code. Note that the `in` operator, when used with dictionaries, iterates over the keys (not the values) in the dictionary.

In [18]:
```
for a in countries:
    print countries[a]["capital"]
```

```
Tashkent
Warsaw
Dar es Salaam
Washington, D.C.
```

That was easy, but we can see that the order is unexpected. As described above, dictionaries store keys in memory according to a hashing function that takes the key value as an input. This means that there is no relationship between the order in which an object is added to a dictionary and the order in which the objects are stored in memory or returned when called. This means that *dictionaries have no inherent ordering, regardless of what order objects print out when called.* If you need ordering information in your data structure, a dictionary is the wrong tool. But, once you stop using locations in lists as a means of identifying associated observations, you will find that there are relatively few occaisions where ordering actually matters and can't be recorded as a value in the dictionary.

If our data was originally in the form of independent lists, it would actually be simpler to collect all of this into a nested structure to begin with, rather than starting with the inner dictionaries.

In [19]:
```
names = ["usa", "poland", "uzbekistan", "tanzania"]
pop = [318968000, 38485779, 30185000, 47400000]
language = ["english", "polish", "uzbek", "swahili"]
gdp = ["$53,001", "$21,118","$4,038","$1,813"]
capital = ["Washington, D.C.", "Warsaw", "Tashkent", "Dar es Salaam"]

countries = {}
for i in range(0, len(names)):
    countries[names[i]] = {
        "population": pop[i],
        "majority language": language[i],
        "GDP per cap": gdp[i],
        "capital": capital[i]
}

countries.items()
```

Out[19]:
```
[('uzbekistan',
  {'GDP per cap': '$4,038',
   'capital': 'Tashkent',
   'majority language': 'uzbek',
   'population': 30185000}),
 ('poland',
  {'GDP per cap': '$21,118',
   'capital': 'Warsaw',
   'majority language': 'polish',
   'population': 38485779}),
 ('tanzania',
  {'GDP per cap': '$1,813',
   'capital': 'Dar es Salaam',
   'majority language': 'swahili',
   'population': 47400000}),
 ('usa',
  {'GDP per cap': '$53,001',
   'capital': 'Washington, D.C.',
   'majority language': 'english',
   'population': 318968000})]
```

This creates an identical dictionary as the one we created before incrementally.

Now, we might like to view just the keys from our outer dictionary, to see which countries we have data for.

In [20]:
```
for key in countries:
    print key

## is equivalent to
#for key in countries.keys():
#    print key
```

```
uzbekistan
poland
tanzania
usa
```

We can print out just the values of the outer dictionary as well.

In [21]:
```
for values in countries.values():
    print values
```

```
{'GDP per cap': '$4,038', 'capital': 'Tashkent', 'majority language': 'uzbek', 'population': 30185000}
{'GDP per cap': '$21,118', 'capital': 'Warsaw', 'majority language': 'polish', 'population': 38485779}
{'GDP per cap': '$1,813', 'capital': 'Dar es Salaam', 'majority language': 'swahili', 'population': 47400000}
{'GDP per cap': '$53,001', 'capital': 'Washington, D.C.', 'majority language': 'english', 'population': 318968000}
```

You can access both keys and dictionaries with `iteritems`

In [22]:
```
for k, v in countries.iteritems():
    print k, v
```

```
uzbekistan {'GDP per cap': '$4,038', 'capital': 'Tashkent', 'majority language': 'uzbek', 'population': 30185000}
poland {'GDP per cap': '$21,118', 'capital': 'Warsaw', 'majority language': 'polish', 'population': 38485779}
tanzania {'GDP per cap': '$1,813', 'capital': 'Dar es Salaam', 'majority language': 'swahili', 'population': 47400000}
usa {'GDP per cap': '$53,001', 'capital': 'Washington, D.C.', 'majority language': 'english', 'population': 318968000}
```

To access an inner `dictionary`, we have to specify which key in the outer dictionary we want to use.

In [23]:
```
countries['usa']
```

Out[23]:
```
{'GDP per cap': '$53,001',
 'capital': 'Washington, D.C.',
 'majority language': 'english',
 'population': 318968000}
```

We can then select keys, values, or both from this inner `dictionary` just as we did with the outer one.

In [24]:
```
countries['usa'].keys()
```

Out[24]:
```
['GDP per cap', 'capital', 'majority language', 'population']
```

In [25]:
```
countries['usa'].values()
```

Out[25]:
```
['$53,001', 'Washington, D.C.', 'english', 318968000]
```

In [26]:
```
countries['usa'].items()
```

Out[26]:
```
[('GDP per cap', '$53,001'),
 ('capital', 'Washington, D.C.'),
 ('majority language', 'english'),
 ('population', 318968000)]
```

Note that `items` returns a list of the key value pairs as tuples, while `iteritems` returns an *iterator* that returns each tuple as you loop over it, just like the trigrams function in `nltk`.

I can grab a specific value by indexing the appropriate keys.

In [27]:
```
countries['usa']['population']
```

Out[27]:
```
318968000
```

I can iterate over the keys in the outer `dictionary` to pull out selected elements from the inner `dictionaries`.

In [28]:
```
for i in countries:
    print i, countries[i]["GDP per cap"]
```

```
uzbekistan $4,038
poland $21,118
tanzania $1,813
usa $53,001
```

`Dicts` are mutable, so I can add elements to every inner dictionary by looping over the keys of the outer one.

In [29]:
```
count = 1
for i in countries:
    countries[i]["hi"] = "hello" + str(count)
    count += 1

for i in countries:
    print i, countries[i]["hi"]
```

```
uzbekistan hello1
poland hello2
tanzania hello3
usa hello4
```

The most natural way to store hierarichal data like this in JSON. Writing nested dictionaries to JSON in python is very, very easy. Reading in JSON files is equally easy with `json.load`. This data is automatically stored as a nested dictionary, so it's a good thing you know all about how to deal with those now!

In [30]:
```
import json

with open("countries.js", "w") as j:
    json.dump(countries, j)
```

For many reasons, I might prefer to use CSV. There's a handy package for writing `dicts` to csv, but it is tricky to get it to write the outer dictionary keys, so I'm first going to add another key to each of the inner dictionaries that duplicates the outer key information.

In [31]:
```
for i in countries:
    countries[i]["name"] = i
```

Now I'll write it to csv using the `DictWriter` class from the `csv` module. First I'll get a list of the keys of one of the inner dictionaries to use as column headers, which goes into the `DictWriter` as the `fieldnames` argument.