[Download notebook]

Don't reinvent the wheel

It's possible to use the standard Python data structures for building blocks for most structures you could imagine.

- `list`
- `tuple`
- `set`
- `dict`

Collections

There is a python module `collections` which contain data structures many find themselves implement often

- `Counter`
- `defaultdict`
- `OrderedDict`
- `deque`
- `namedtuple()`

Hashable objects

A *hashable collection* in Python is any data structure which stores data in such a way that it uses a hash table for data accession.

To be placeable in a hashable collection an object need to implement `__hash__()`, which should return an `int`, which should also never change value during the lifetime of that object. Otherwise the object would be stored in the wrong place after it is changed.

In addition, the objects need to implement `__eq__()` or `__cmp__()` so it can be evaluated whether two colliding objects are different from each other or if it is the same object.

Objects satisfying these two conditions are called *hashable*.

All the immutable built in Python data types are hashable (i.e. the primitive data types, as well as `str` and `tuple`).

`Counter`

Counting the number of occurrences of items is very common, and can in Python be implemented with a dictionary where the keys are the objects being counted, and the values the count.

An efficient implementation of this, along with many convenient features, is avalable in `collections.Counter`.

In [1]:
```
from collections import Counter
```

In [2]:
```
c = Counter()
```

In [3]:
```
c
```

Out[3]:
```
Counter()
```

In [4]:
```
c["a"]
```

Out[4]:
```
0
```

In [7]:
```
c.update("a")
```

In [8]:
```
c
```

Out[8]:
```
Counter({'a': 2})
```

We can also instantiate the `Counter` by passing it a container of hashable objects.

In [9]:
```
c = Counter("A quite short but in other ways normal English
sentence".lower().replace(" ", ""))
```

In [10]:
```
c
```

Out[10]:
```
Counter({'e': 6, 'n': 5, 't': 5, 's': 4, 'a': 3, 'i': 3, 'h': 3, 'o': 3, 'r':
3, 'l': 2, 'u': 2, 'c': 1, 'b': 1, 'g': 1, 'm': 1, 'q': 1, 'w': 1, 'y': 1})
```

In [11]:
```
c.most_common(5)
```

Out[11]:
```
[('e', 6), ('n', 5), ('t', 5), ('s', 4), ('a', 3)]
```

In [13]:
```
c.
```

Out[13]:
```
6
```

`Counters` supports mathematical operations which makes them behave like multisets, for which there are some algorithmic uses.

defaultdict

Say that we want to implement the `Counter` functionality for a normal dict ourselves. A function taking a container of hashables and returning a `dict` of counts would look like this.

In [14]:
```
def count_items(collection):
    counts = dict()
    for item in iter(collection):
        if item in counts:
            counts[item] += 1
        else:
            counts[item] = 1

    return counts
```

In [15]:
```
count_items("test")
```

Out[15]:
```
{'e': 1, 's': 1, 't': 2}
```

This recipe of doing things to items in a `dict` in case the exist, or adding them if they don't, is *extremely* common. So in the interest of not reinventing the wheel there is `collections.defauldict`.

A `defaultdict` is instantiated with a *factory* for objects which will be added to the dictionary if we try to access an item which is not there.

The term *factory* refers to an object which creates other objects.

For example, `int` is a `type` object, which produces `ints` when called.

In [17]:
```
type(int)
```

Out[17]:
```
type
```

In [18]:
```
type(int())
```

Out[18]:
```
int
```

In [20]:
```
int()
```

Out[20]: 0

Normally types are used as factories, but it can actually be any function which returns something.

In [16]:
```
from collections import defaultdict

A = defaultdict(int)
A
```

Out[16]:
```
defaultdict(<type 'int'>, {})
```

In [21]:
```
A["test"]
```

Out[21]: 0

In [22]:
```
A
```

Out[22]:
```
defaultdict(<type 'int'>, {'test': 0})
```

In [23]:
```
B = defaultdict(list)
B["one"].append(5)
B
```

Out[23]:
```
defaultdict(<type 'list'>, {'one': [5]})
```

In [24]:
```
B["one"].append(3)
```

In [25]:
```
B
```

Out[25]:
```
defaultdict(<type 'list'>, {'one': [5, 3]})
```

In [26]:
```
C = defaultdict(lambda: defaultdict(int))
C["outer"]["inner"] += 7
C
```

Out[26]:
```
defaultdict(<function <lambda> at 0x10fd3ecf8>, {'outer': defaultdict(<type
'int'>, {'inner': 7})})
```

In [27]:
```
def defaultdict_factory():
    return defaultdict(defaultdict_factory)

D = defaultdict(defaultdict_factory)
D[0][0][0][0][0] = "Inside a dict of dicts of ... of dicts (dynamically)"
D
```

Out[27]:
```
defaultdict(<function defaultdict_factory at 0x10fd3ed70>, {0:
defaultdict(<function defaultdict_factory at 0x10fd3ed70>, {0:
defaultdict(<function defaultdict_factory at 0x10fd3ed70>, {0:
defaultdict(<function defaultdict_factory at 0x10fd3ed70>, {0:
defaultdict(<function defaultdict_factory at 0x10fd3ed70>, {0: 'Inside a dict
of dicts of ... of dicts (dynamically)'})})})})})
```

OrderedDict

Since `dicts` are implemented to give fast access to the items, the data is stored in a way that makes that quick. But sometimes you might want it sorted so it is possible to iterate over it in a predictable way.

In [28]:
```
d = {}
```

```
for k, v in zip(["a", "b", "g", "d"], [u"α", u"β", u"γ", u"δ"]):
    d[k] = v

d
```

Out[28]:

```
{'a': u'\u03b1', 'b': u'\u03b2', 'd': u'\u03b4', 'g': u'\u03b3'}
```

In [29]:

```
from collections import OrderedDict

od = OrderedDict()

for k, v in zip(["a", "b", "g", "d"], [u"α", u"β", u"γ", u"δ"]):
    od[k] = v

od
```

Out[29]:

```
OrderedDict([('a', u'\u03b1'), ('b', u'\u03b2'), ('g', u'\u03b3'), ('d',
u'\u03b4')])
```

In [30]:

```
"a", u"a", r"a"
```

Out[30]:

```
('a', u'a', 'a')
```

deque

In Python, `lists` are optimized for operations on a fixed size (even though the sizes of the *elements* might vary, unlike with an `array`).

In [31]:

```
a = range(5)
a
```

Out[31]:

```
[0, 1, 2, 3, 4]
```

When treating a `list` like a stack (where the right end is the top), everything is fine

In [32]:

```
a.pop()
```

Out[32]:

```
4
```

In [33]:

```
a
```

Out[33]:

```
[0, 1, 2, 3]
```

In [34]:

```
a.append(6)
a.append(7)
a
```

Out[34]:

```
[0, 1, 2, 3, 6, 7]
```

In [35]:

```
a.pop()
```

Out[35]:

```
7
```

But when working from the left end will cause `list` performance to be $O(n)$ rather than $O(1)$ due to memory movement.

In [36]:

```
a.pop(0)
```

Out[36]:

```
0
```

In [37]:
```
a.insert(0, 8)
a.insert(0, 9)
a
```

Out[37]:
```
[9, 8, 1, 2, 3, 6]
```

A *deque* ("double-ended queue") is a structure which gives $O(1)$ time for both insertion and accession at both ends.

What you're giving up is the ability to slice.

Accession by indexes is near $O(1)$ close to the ends, but goes towards $O(n)$ closer to the middle.

One can view a `deque` as a circular `list` where we have a reference to a particular section of the circle.

By essentially popping from one end and inserting to the other, the `deque` can be rotated. This is something that can be exploited when writing algorithms where we can take advantage of data locality for portions of the runtime.

In [38]:
```
from collections import deque
```

In [39]:
```
%%timeit
# Right append - pop list

l = []
for z in xrange(1000):
    l.append(z)

for y in xrange(len(l)):
    l.pop()
```

```
1000 loops, best of 3: 434 us per loop
```

In [40]:
```
%%timeit
# Right append - pop deque

q = deque()
for z in xrange(1000):
   q.append(z)

while True:
    try:
        q.pop()
    except:
        break
```

```
1000 loops, best of 3: 455 us per loop
```

In [41]:
```
%%timeit
# Left append - pop list

l = []
for z in xrange(1000):
    l.insert(0, z)

for y in xrange(len(l)):
    l.pop(0)
```

```
100 loops, best of 3: 1.52 ms per loop
```

In [42]:
```
%%timeit
# Left append - pop deque

q = deque()
for z in xrange(1000):
   q.appendleft(z)
```

```
while True:
    try:
        q.popleft()
    except:
        break
```

```
1000 loops, best of 3: 434 us per loop
```

In [43]:
```
a
```

Out[43]:
```
[9, 8, 1, 2, 3, 6]
```

In [44]:
```
b = deque(a)
b
```

Out[44]:
```
deque([9, 8, 1, 2, 3, 6])
```

In [45]:
```
b.rotate(2)
b
```

Out[45]:
```
deque([3, 6, 9, 8, 1, 2])
```

`namedtuple()`

A simple factory for creating subclasses of `tuple` with predefined fields which can be accesses like attributes

In [46]:
```
from collections import namedtuple
```

In [47]:
```
Point = namedtuple("Point", "x y z")
```

In [48]:
```
type(Point)
```

Out[48]:
```
type
```

In [49]:
```
p1 = Point(3,4,4)
```

In [50]:
```
p1
```

Out[50]:
```
Point(x=3, y=4, z=4)
```

In [51]:
```
Point(3,4,53,3,5)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-51-d2b517c9472f> in <module>()
----> 1 Point(3,4,53,3,5)

TypeError: __new__() takes exactly 4 arguments (6 given)
```

In [54]:
```
p1[0]
```

Out[54]:
```
3
```

In [55]:
```
p1.x
```

Out[55]:
```
3
```

The *primary* reason to use a namedtuple instead of creating a class is to not have to type too much before one knows all the functionality of a class is needed.

However, `tuples` are much simpler than normal (unoptimized) classes, and will therefore perform better.

In [56]:
```
class ClassPoint(object):
    def __init__(self, x, y, z):
        self.x = x
        self.y = y
        self.z = z
```

In [57]:
```
%%timeit
# namedtuples
l = range(10000)
for i in xrange(10000):
    l[i] = Point(i, 2 * i, 3 * i)
```
```
100 loops, best of 3: 12.5 ms per loop
```

In [58]:
```
%%timeit
# objects
l = range(10000)
for i in xrange(10000):
    l[i] = ClassPoint(i, 2 * i, 3 * i)
```
```
10 loops, best of 3: 18.1 ms per loop
```

In [60]:
```
a = [1,2,3,4]
```

In [61]:
```
a.insert(2, 0)
```

In [63]:
```
aa = deque(a)
```

In [64]:
```
b = deque()
```

In [65]:
```
b.append(aa)
```

In [67]:
```
b.append(aa)
```

In [68]:
```
b
```

Out[68]:
```
deque([deque([1, 2, 0, 3, 4]), deque([1, 2, 0, 3, 4])])
```

SciPy

`pip install scipy`

http://scipy.org/

While NumPy provides a powerful array language interface to Python, SciPy provides functionality which uses these arrays for numerical calculations.

If you're thinking of writing numerical code, it's a good idea to first check the SciPy documentation; there usually is a good implementation of what you need already.

SciPy is divided in to submodules based on the kind of problem they aim to solve. Some highlights:

- `scipy.linalg`
- `scipy.sparse`
- `scipy.integrate`
- `scipy.optimize`

- scipy.interpolate
- scipy.fftpack
- scipy.signal
- scipy.stats

The SciPy cookbook has plenty of nice examples on how to use the different components.

http://scipy.org/Cookbook

In [70]:
```
from scipy import interpolate
```

In [71]:
```
help(interpolate.PiecewisePolynomial)
```

```
Help on class PiecewisePolynomial in module scipy.interpolate.polyint:

class PiecewisePolynomial(__builtin__.object)
 |  Piecewise polynomial curve specified by points and derivatives
 |  
 |  This class represents a curve that is a piecewise polynomial. It
 |  passes through a list of points and has specified derivatives at
 |  each point. The degree of the polynomial may very from segment to
 |  segment, as may the number of derivatives available. The degree
 |  should not exceed about thirty.
 |  
 |  Appending points to the end of the curve is efficient.
 |  
 |  Methods defined here:
 |  
 |  __call__(self, x)
 |      Evaluate the piecewise polynomial
 |      
 |      Parameters
 |      ----------
 |      x : scalar or array-like of length N
 |      
 |      Returns
 |      -------
 |      y : scalar or array-like of length R or length N or N by R
 |  
 |  __init__(self, xi, yi, orders=None, direction=None)
 |      Construct a piecewise polynomial
 |      
 |      Parameters
 |      ----------
 |      xi : array-like of length N
 |          a sorted list of x-coordinates
 |      yi : list of lists of length N
 |          yi[i] is the list of derivatives known at xi[i]
 |      orders : list of integers, or integer
 |          a list of polynomial orders, or a single universal order
 |      direction : {None, 1, -1}
 |          indicates whether the xi are increasing or decreasing
 |          +1 indicates increasing
 |          -1 indicates decreasing
 |          None indicates that it should be deduced from the first two xi
 |      
 |      Notes
 |      -----
 |      If orders is None, or orders[i] is None, then the degree of the
 |      polynomial segment is exactly the degree required to match all i
 |      available derivatives at both endpoints. If orders[i] is not None,
 |      then some derivatives will be ignored. The code will try to use an
 |      equal number of derivatives from each end; if the total number of
 |      derivatives needed is odd, it will prefer the rightmost endpoint. If
 |      not enough derivatives are available, an exception is raised.
```

```
 |
 |  append(self, xi, yi, order=None)
 |      Append a single point with derivatives to the PiecewisePolynomial
 |
 |      Parameters
 |      ----------
 |      xi : float
 |
 |      yi : array_like
 |          yi is the list of derivatives known at xi
 |
 |      order : integer or None
 |          a polynomial order, or instructions to use the highest
 |          possible order
 |
 |  derivative(self, x, der)
 |      Evaluate a derivative of the piecewise polynomial
 |
 |      Parameters
 |      ----------
 |      x : scalar or array_like of length N
 |
 |      der : integer
 |          which single derivative to extract
 |
 |      Returns
 |      -------
 |      y : scalar or array_like of length R or length N or N by R
 |
 |      Notes
 |      -----
 |      This currently computes (using self.derivatives()) all derivatives
 |      of the curve segment containing each x but returns only one.
 |
 |  derivatives(self, x, der)
 |      Evaluate a derivative of the piecewise polynomial
 |
 |      Parameters
 |      ----------
 |      x : scalar or array_like of length N
 |
 |      der : integer
 |          how many derivatives (including the function value as
 |          0th derivative) to extract
 |
 |      Returns
 |      -------
 |      y : array_like of shape der by R or der by N or der by N by R
 |
 |  extend(self, xi, yi, orders=None)
 |      Extend the PiecewisePolynomial by a list of points
 |
 |      Parameters
 |      ----------
 |      xi : array_like of length N1
 |          a sorted list of x-coordinates
 |      yi : list of lists of length N1
 |          yi[i] is the list of derivatives known at xi[i]
 |      orders : list of integers, or integer
 |          a list of polynomial orders, or a single universal order
 |      direction : {None, 1, -1}
 |          indicates whether the xi are increasing or decreasing
 |          +1 indicates increasing
 |          -1 indicates decreasing
 |          None indicates that it should be deduced from the first two xi
```

```
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors defined here:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)
```

In general, methods in SciPy can take an 'array-like' which will be converted to an `np.ndarray`, and then used in the SciPy modules.

SciKits

http://scikits.appspot.com/

SciKits (SciPy Toolkits) are packages which builds upon SciPy, but are not included with the SciPy package for any of these reasons:

- Package uses a license which is not compatible with the SciPy license
- Package is too specialized to be worth distributing with SciPy (e.g. `hydroclimpy`)
- Package is undergoing rapid development and can change in ways that might break backward compatibility

Examples of scikits you hear a lot about

- `statsmodels`
- `scikit-image`
- `scikit-learn`

Pandas

`pip install pandas`

http://pandas.pydata.org/

Most notably provides two data structures which builds upon NumPy arrays:

`Series` for 1-dimensional data

`DataFrame` for 2-dimensional data

- tabular data
- time series
- panel data

Combined with IPython, (and in particular IPython Notebook), Pandas and the numerical packages becomes great for exploring data.

`Series`

One dimensional labeled array, implemented as subclass of `ndarray`.

While a `ndarray` can access elements and slices with numerical indexes, a `Series` is indexed by a list of labels for each element.

In [72]:
```
from pandas import Series
```

In [73]:
```
a = np.random.randn(5)
a
```

Out[73]:
```
array([ 1.89454593,  0.90264556, -1.14669791,  0.46067578,  0.32107422])
```

In [74]:
```
Series(a)
```

Out[74]:
```
0    1.894546
1    0.902646
2   -1.146698
3    0.460676
4    0.321074
```

In [75]:
```
s = Series(a, index=['a', 'b', 'c', 'd', 'e'])
s
```

Out[75]:
```
a    1.894546
b    0.902646
c   -1.146698
d    0.460676
e    0.321074
```

In [76]:
```
a[1:4]
```

Out[76]:
```
array([ 0.90264556, -1.14669791,  0.46067578])
```

In [77]:
```
s['b':'d']
```

Out[77]:
```
b    0.902646
c   -1.146698
d    0.460676
```

In many ways a `Series` can be seen as a cross between an `ndarray` and a `dict`, and a very logical way of creating a `Series` is by passing it a `dict`.

In [78]:
```
d = {'a': 0., 'b': 1., 'c': 2.}
s1 = Series(d)
s1
```

Out[78]:
```
a    0
b    1
c    2
```

In [79]:
```
s1.index
```

Out[79]:
```
Index([a, b, c], dtype=object)
```

(Note that the `list` we passed have been converted to an `Index` object.)

In most ways a `Series` can be used just as an `ndarray`, however, when doing operations with different series, the data will automatically be aligned by the index labels

In [80]:
```
# ndarray
a[1:] + a[:-1]
```

Out[80]:
```
array([ 2.79719149, -0.24405236, -0.68602213,  0.78175    ])
```

In [81]:
```
# Series
s[1:] + s[:-1]
```

Out[81]:
```
a         NaN
b    1.805291
c   -2.293396
d    0.921352
e         NaN
```

To help keep track of `Series`, they can be given names.

In [82]:
```
s = Series(a, index=['a', 'b', 'c', 'd', 'e'], name="Random example data")
s
```

Out[82]:
```
a    1.894546
b    0.902646
c   -1.146698
d    0.460676
e    0.321074
Name: Random example data
```

`DataFrame`

Two-dimensional labeled data structure, most intuitively thought of as a **table**.

In [83]:
```
from pandas import DataFrame
```

A `DataFrame` can be instantiated in many ways, but the most intuitive is when constructed from a `dict` of `Series`.

In [84]:
```
d = {'one' : Series([1., 2., 3.], index=['a', 'b', 'c']), \
     'two' : Series([1., 2., 3., 4.], index=['a', 'b', 'c', 'd'])}

df = DataFrame(d)
df
```

Out[84]:

	one	two
a	1	1
b	2	2
c	3	3
d	NaN	4

In [85]:
```
df.index
```

Out[85]:
```
Index([a, b, c, d], dtype=object)
```

In [86]:
```
df.columns
```

Out[86]:
```
Index([one, two], dtype=object)
```

In [87]:
```
df1 = DataFrame(randn(10, 4), columns=['A', 'B', 'C', 'D'])
df2 = DataFrame(randn(7, 3), columns=['A', 'B', 'C'])
```

Accessing elements directly from a `DataFrame` will give you a `Series`

In [88]:
```
df1["A"]
```

Out[88]:
```
0   -0.769256
1   -0.392941
2   -0.036154
3   -0.042374
4    1.058781
5   -0.982873
6    0.045609
7   -1.608038
8    0.670419
```

```
9    1.020733
Name: A
```

Slicing directly however only works on rows

In [89]:
```
df1[2:4]
```

Out[89]:

	A	B	C	D
2	-0.036154	-0.937833	0.776257	0.875263
3	-0.042374	-0.715209	-0.471679	-0.189201

To slice in any direction, use the `ix` attribute of a `DataFrame`.

In [90]:
```
df1.ix[2:,"B":"D"]
```

Out[90]:

	B	C	D
2	-0.937833	0.776257	0.875263
3	-0.715209	-0.471679	-0.189201
4	-0.954273	-0.575008	-0.881008
5	0.314320	0.351847	0.034518
6	-0.758584	1.170517	0.234722
7	0.096029	0.974100	-0.860451
8	0.173108	-0.254731	0.968456
9	0.030799	0.231640	0.212490

To get a **row** as series, use the `xs` method

In [91]:
```
df1.xs(5)
```

Out[91]:
```
A   -0.982873
B    0.314320
C    0.351847
D    0.034518
Name: 5
```

In [92]:
```
df1 + df2
```

Out[92]:

	A	B	C	D
0	-1.390004	1.357910	-0.212756	NaN
1	0.579743	-2.053898	0.110167	NaN
2	0.291970	-1.828201	1.277833	NaN
3	0.151515	0.590763	-2.604155	NaN
4	0.466383	-0.067969	-0.152371	NaN
5	-1.005964	0.394166	-0.380423	NaN
6	-1.342940	-1.582353	1.359328	NaN
7	NaN	NaN	NaN	NaN
8	NaN	NaN	NaN	NaN

9	NaN	NaN	NaN	NaN

In [93]:
```
df1 * 20
```

Out[93]:

	A	B	C	D
0	-15.385119	2.827629	-14.666131	-24.950579
1	-7.858822	-21.900526	7.637977	4.545096
2	-0.723078	-18.756662	15.525133	17.505269
3	-0.847483	-14.304187	-9.433586	-3.784014
4	21.175615	-19.085451	-11.500155	-17.620163
5	-19.657465	6.286394	7.036946	0.690362
6	0.912172	-15.171680	23.410348	4.694435
7	-32.160761	1.920575	19.481998	-17.209029
8	13.408376	3.462153	-5.094613	19.369123
9	20.414662	0.615983	4.632805	4.249790

Broadcasting with Series is done row wise:

In [94]:
```
df1 + df2.xs(2)
```

Out[94]:

	A	B	C	D
0	-0.441132	-0.748987	-0.231730	NaN
1	-0.064817	-1.985395	0.883475	NaN
2	0.291970	-1.828201	1.277833	NaN
3	0.285750	-1.605578	0.029897	NaN
4	1.386905	-1.844641	-0.073431	NaN
5	-0.654749	-0.576049	0.853424	NaN
6	0.373733	-1.648952	1.672094	NaN
7	-1.279914	-0.794340	1.475676	NaN
8	0.998543	-0.717261	0.246846	NaN
9	1.348857	-0.859569	0.733217	NaN

For a `DataFrame`, each **column** must have the same type (unlike a 2d `ndarray`).

When all columns have numerical values, NumPy functions can be used on the `DataFrame` just as if it was a 2d `ndarray`.

In [95]:
```
np.exp(df1)
```

Out[95]:

	A	B	C	D
0	0.463358	1.151864	0.480318	0.287214
1	0.675069	0.334531	1.465064	1.255150
2	0.964492	0.391475	2.173322	2.399507
3	0.958511	0.489090	0.623954	0.827620

4	2.882854	0.385092	0.562701	0.414365
5	0.374234	1.369327	1.421691	1.035121
6	1.046665	0.468329	3.223660	1.264557
7	0.200280	1.100791	2.648782	0.422971
8	1.955056	1.188994	0.775125	2.633875
9	2.775229	1.031278	1.260666	1.236753

This means `DataFrames` works almost seemlessly with SciPy or SciKits. In particular developers of statistics SciKits have started putting a lot of effort in to making their packages as Pandas compatible as possible.

It should be noted that Pandas also has a `Panel` data structure, which is a 3-dimensional structure which can be viewed conceptually as a "Series of DataFrames".

In addition to this there are experimental data structures `Panel4D` and `PanelND`.

Assignment for Session 4

Github API

Github has a very extensive RESTful API (a web service). For example, to get information about members of an organization, one can do this

In [96]:
```
import requests
```

In [97]:
```
with open("secret") as secret:
    password = secret.read().strip()
```

```
---------------------------------------------------------------------------
IOError                                   Traceback (most recent call last)
<ipython-input-97-ce01dca0116a> in <module>()
----> 1 with open("secret") as secret:
      2     password = secret.read().strip()

IOError: [Errno 2] No such file or directory: 'secret'
```

In [98]:
```
users = requests.get("https://api.github.com/orgs/pythonkurs/members",
auth=("vals", password))
```

NEVER EVER PUT CREDENTIALS AS TEXT IN SCRIPTS!

In [99]:
```
users_data = users.json()
```

In [100]:
```
len(users_data)
```

Out[100]: 48

In [101]:
```
users_data[0]
```

Out[101]:
```
{u'avatar_url':
u'https://secure.gravatar.com/avatar/e2a0accefc4b4298a35e76da78a0f52c?
d=https://a248.e.akamai.net/assets.github.com%2Fimages%2Fgravatars%2Fgravatar-
user-420.png',
 u'events_url': u'https://api.github.com/users/alneberg/events{/privacy}',
 u'followers_url': u'https://api.github.com/users/alneberg/followers',
 u'following_url': u'https://api.github.com/users/alneberg/following',
 u'gists_url': u'https://api.github.com/users/alneberg/gists{/gist_id}',
 u'gravatar_id': u'e2a0accefc4b4298a35e76da78a0f52c',
```

```
 u'id': 1250075,
 u'login': u'alneberg',
 u'organizations_url': u'https://api.github.com/users/alneberg/orgs',
 u'received_events_url':
u'https://api.github.com/users/alneberg/received_events',
 u'repos_url': u'https://api.github.com/users/alneberg/repos',
 u'starred_url': u'https://api.github.com/users/alneberg/starred{/owner}
{/repo}',
 u'subscriptions_url':
u'https://api.github.com/users/alneberg/subscriptions',
 u'type': u'User',
 u'url': u'https://api.github.com/users/alneberg'}
```


The API is documented very clearly and with lots of examples at http://developer.github.com/

Note in particular that URI's to related resources are given with keys of the form `*_url`

There are (at the moment of writing) 28 repositories in the pythonkurs organization.

1.

Use the Github API to get the commit history of each repository in the organization in to a DataFrame, where columns are repositories and rows are commits indexed by time. The content should be the commit message.

Depending on strategy for getting the commit times, you might want to use the third party package `dateutils` to parse the "date" values. It is installable by

```
pip install dateutil
```

In [102]:
```
from dateutil import parser
```

In [103]:
```
parser.parse("2013-02-04T16:58:05Z")
```

Out[103]:
```
datetime.datetime(2013, 2, 4, 16, 58, 5, tzinfo=tzutc())
```

For clarification, this could be how one `Series` in the `DataFrame` would look like

In [104]:
```
import datetime

base = datetime.datetime.now()
date_list = [base - datetime.timedelta(days=x) for x in range(5)]

s = Series(["A commit message"] * 5, index=date_list, name="A repo")
s
```

Out[104]:
```
2013-02-06 15:39:58.265767    A commit message
2013-02-05 15:39:58.265767    A commit message
2013-02-04 15:39:58.265767    A commit message
2013-02-03 15:39:58.265767    A commit message
2013-02-02 15:39:58.265767    A commit message
Name: A repo
```

In [105]:
```
DataFrame(s)
```

Out[105]:

	A repo
2013-02-06 15:39:58.265767	A commit message
2013-02-05 15:39:58.265767	A commit message
2013-02-04 15:39:58.265767	A commit message
2013-02-03 15:39:58.265767	A commit message
2013-02-02 15:39:58.265767	A commit message

2013-02-02 [illegible]	A commit message

Put a function which returns the specified `DataFrame` in a submodule.

NOTE: DO NOT PUT ANY CREDENTIALS IN ANY FILES ADDED TO THE REPO. You need to figure out a good way of handling the authentication. (The *best* way is to use OAuth, but it is rather complicated.) One solution could for example be to make the function take credentials as an argument.

2.

Use the `DataFrame` to figure out **what is the most common weekday and hour of a day to commit to a course repo**.

To do this, you might want to do some interactive exploring of the data. This can be done in the standard Python interpreter, but your life might become a lot easier if you use IPython. IPython can be installed by

```
pip install ipython
```

and can then be started by running `ipython` in stead of `python`.

You might also want to try the IPython Notebook. In that case you also need to install a couple of dependencies by doing

```
pip install tornado
pip install pyzmq
```

You *should* then be able to start the notebook by running

```
ipython notebook --pylab=inline
```

If it doesn't work, don't worry about it, just use the normal IPython.

When you have figured out a way of giving a value for the most common weekday and hour of a day, write a function in a `session4` submodule which takes a `DataFrame` as argument and returns the weekday and hour of a day.

3.

When you have learned how to get data from the Github API, spend some time playing around with data you can get and try to figure out something else about the pythonkurs organization.

No need to write functions or so for this assignment, just explore. Then write a *very* short plain text file in the root of your repository called **`pythonkurs_organization.txt`** with some information about the data that you found interesting.

By "*very* short", we mean **no longer than 150 words**. Putting more than 150 words in the text file will count as failing the assignment.

Of course, "interesting" is subjective. The important part is showing that you explored the data some.

After this assignment, your repository should contain these files:

```
surname/
    surname/
        __init__.py
        session2.py
        session3.py
        session4.py    <- new

    scripts/
        getting_data.py
        check_repo.py

    README.md
    setup.py
    pythonkurs_organization.txt    <- new
```

(As well as other files, but these are the ones we have talked about.)

In [108]:
```
df1.shape
```

Out[108]: (10, 4)

In [109]:
```
df.ix[:7, :3].shape
```

Out[109]: (4, 2)