Case Studies and Applications

July 19, 2015

1 Financial and Economic Data Applications

In this final lecture, we will look at a number of useful applications of the techniques discussed in previous weeks. These all come from the worlds of finance and economics, mostly because of the abundance of data. We will also be using Wes McKinney's template `.ipynb` file as a skeleton for the lectures.

Throughout, we will make use of some technical jargon. Here are frequent terms that you should be familiar with:

- *cross-sectional* data: data that comprise a fixed point in time (for example, the current closing price of publicly traded companies on April 1 2015).
- *panel* data: multi-dimensional data, which frequently involve time series measurements but can also be cross-sectional.
- *futures contract*: a financial instrument in which two parties agree to the sale of a distinct instrument (such as corn, oil, or a stock) sometime into the future, thereby deriving its value from the value of an underlying asset. As such, it is a form of *derivative* contract.

```
In [1]: from __future__ import division
        from pandas import Series, DataFrame
        import pandas as pd
        from numpy.random import randn
        import numpy as np
        pd.options.display.max_rows = 12
        np.set_printoptions(precision=4, suppress=True)
        import matplotlib.pyplot as plt
```

```
In [2]: %matplotlib inline
```

1.1 Data munging topics

In previous lectures we have discussed a number of useful tools that are present within the Python data analysis ecosystem for data munging. Here we will overview a few of these tools in action.

1.1.1 Time series and cross-section alignment

Suppose you are given two different financial datasets on which you hope to perform a reasonable analysis. More likely than not, the series may have indices that don't line up perfectly, or (if the data is stored into a `DataFrame`) might have columns or row labels that don't match.

This is a terribly frequent and frustrating problem in data analysis, so much so that it has its own name: the *data alignment problem*. In `Pandas`, basic arithmetic operations between data sets performs automatic alignment. For example, let us consider the following datasets.

```
In [3]: close_px = pd.read_csv('ch11/stock_px.csv', parse_dates=True, index_col=0)
        volume = pd.read_csv('ch11/volume.csv', parse_dates=True, index_col=0)
        prices = close_px.ix['2011-09-05':'2011-09-14', ['AAPL', 'JNJ', 'SPX', 'XOM']]
        volume = volume.ix['2011-09-05':'2011-09-12', ['AAPL', 'JNJ', 'XOM']]
```

```
In [4]: prices

Out[4]:               AAPL    JNJ      SPX    XOM
        2011-09-06  379.74  64.64  1165.24  71.15
        2011-09-07  383.93  65.43  1198.62  73.65
        2011-09-08  384.14  64.95  1185.90  72.82
        2011-09-09  377.48  63.64  1154.23  71.01
        2011-09-12  379.94  63.59  1162.27  71.84
        2011-09-13  384.62  63.61  1172.87  71.65
        2011-09-14  389.30  63.73  1188.68  72.64
```

```
In [5]: volume

Out[5]:                 AAPL       JNJ       XOM
        2011-09-06  18173500  15848300  25416300
        2011-09-07  12492000  10759700  23108400
        2011-09-08  14839800  15551500  22434800
        2011-09-09  20171900  17008200  27969100
        2011-09-12  16697300  13448200  26205800
```

One useful metric in the financial analysis of stock prices is the *volume-weighted average price*, or VWAP, of a stock. This metric weights the price of a stock at any given time by the amount of trades are occurring at that time. The idea is that high-volume trades give a better indication of the perceptions of institutional investors, who presumably have expert understanding, of the value of the stock.

Since `Pandas` aligns the data automatically and excludes missing data in functions like `sum`, we can express this concisely as:

```
In [6]: prices * volume

Out[6]:                   AAPL         JNJ  SPX         XOM
        2011-09-06  6901204890  1024434112  NaN  1808369745
        2011-09-07  4796053560   704007171  NaN  1701933660
        2011-09-08  5700560772  1010069925  NaN  1633702136
        2011-09-09  7614488812  1082401848  NaN  1986085791
        2011-09-12  6343972162   855171038  NaN  1882624672
        2011-09-13         NaN         NaN  NaN         NaN
        2011-09-14         NaN         NaN  NaN         NaN
```

```
In [7]: vwap = (prices * volume).sum() / volume.sum()

In [8]: vwap

Out[8]: AAPL    380.655181
        JNJ      64.394769
        SPX            NaN
        XOM      72.024288
        dtype: float64
```

Because no volume data is given for the SPX exchange-traded fund, the VWAP value for the asset is consequently absent. We can remove it from the above series by a simple call to `dropna`.

```
In [9]: vwap.dropna()

Out[9]: AAPL    380.655181
        JNJ      64.394769
        XOM      72.024288
        dtype: float64
```

Manual alignment can be achieved by using the `align` method built-in to `DataFrame` objects.

```
In [10]: prices.align(volume, join='inner')

Out[10]: (              AAPL    JNJ    XOM
          2011-09-06  379.74  64.64  71.15
          2011-09-07  383.93  65.43  73.65
          2011-09-08  384.14  64.95  72.82
          2011-09-09  377.48  63.64  71.01
          2011-09-12  379.94  63.59  71.84,                  AAPL       JNJ       XOM
          2011-09-06  18173500  15848300  25416300
          2011-09-07  12492000  10759700  23108400
          2011-09-08  14839800  15551500  22434800
          2011-09-09  20171900  17008200  27969100
          2011-09-12  16697300  13448200  26205800)
```

You can also build `DataFrame` objects with `Series` objects that may potentially have different individual shapes without a hitch, because of `Pandas` automatic alignment.

```
In [11]: s1 = Series(range(3), index=['a', 'b', 'c'])
         s2 = Series(range(4), index=['d', 'b', 'c', 'e'])
         s3 = Series(range(3), index=['f', 'a', 'c'])
         DataFrame({'one': s1, 'two': s2, 'three': s3})

Out[11]:    one  three  two
         a    0      1  NaN
         b    1    NaN    1
         c    2      2    2
         d  NaN    NaN    0
         e  NaN    NaN    3
         f  NaN      0  NaN
```

Again, `NaN` is assigned to values for which the original `Series` do not cover respectively.
You can specify explicitly the index of the result, discarding the rest:

```
In [12]: DataFrame({'one': s1, 'two': s2, 'three': s3}, index=list('face'))

Out[12]:    one  three  two
         f  NaN      0  NaN
         a    0      1  NaN
         c    2      2    2
         e  NaN    NaN    3
```

1.1.2 Operations with time series of different frequencies

The Federal Reserve publishes new GDP data every quarter, but only publishes inflation data annually. Publicly listed firms are required to provide income and balance sheet data quarterly, but it need not happen at the same day for every company. These types of problems for data analysts fall under the category of time series frequency problems, and `Pandas` provides a number of techniques for solving them.

Suppose you have a time series that contains data compiled weekly (on Wednesdays):

```
In [13]: ts1 = Series(np.random.randn(3),
                      index=pd.date_range('2012-6-13', periods=3, freq='W-WED'))
         ts1
```

```
Out[13]: 2012-06-13    2.532694
         2012-06-20   -0.288329
         2012-06-27    1.191570
         Freq: W-WED, dtype: float64
```

In certain circumstances, it might be useful to resample the data to different frequencies. For example, if you need to resample the data so that an entry is provided for every business day in the period, you can employ:

```
In [14]: ts1.resample('B')
```

```
Out[14]: 2012-06-13    2.532694
         2012-06-14         NaN
         2012-06-15         NaN
         2012-06-18         NaN
         2012-06-19         NaN
         2012-06-20   -0.288329
         2012-06-21         NaN
         2012-06-22         NaN
         2012-06-25         NaN
         2012-06-26         NaN
         2012-06-27    1.191570
         Freq: B, dtype: float64
```

Notice here that because no new information is provided, all of the new dates are simply left `NaN`. If you want to fill these gaps with prevous data, you can apply various fillers with the `fill_method` parameter specified.

```
In [15]: ts1.resample('B', fill_method='ffill')
```

```
Out[15]: 2012-06-13    2.532694
         2012-06-14    2.532694
         2012-06-15    2.532694
         2012-06-18    2.532694
         2012-06-19    2.532694
         2012-06-20   -0.288329
         2012-06-21   -0.288329
         2012-06-22   -0.288329
         2012-06-25   -0.288329
         2012-06-26   -0.288329
         2012-06-27    1.191570
         Freq: B, dtype: float64
```

This remedy is an elegant solution to upsampling from lower frequency data to higher frequency data, but another class of frequency problems involve irregular time series data, in which the above methods will not work as neatly. Consider the following data set containing irregularly sampled data:

```
In [16]: dates = pd.DatetimeIndex(['2012-6-12', '2012-6-17', '2012-6-18',
                                   '2012-6-21', '2012-6-22', '2012-6-29'])
         ts2 = Series(np.random.randn(6), index=dates)
         ts2
```

```
Out[16]: 2012-06-12    1.175068
         2012-06-17    0.595031
         2012-06-18    0.106952
         2012-06-21   -0.165089
```

```
2012-06-22   -0.187681
2012-06-29    0.312908
dtype: float64
```

If you want to add the "as of" values in `ts1` to `ts2`, one option would be to resample both to a regular frequency and then add, but if you want to maintain the date index in `ts2`, using `reindex` is a more precise solution:

```
In [17]: ts1.reindex(ts2.index, method='ffill')
```

```
Out[17]: 2012-06-12         NaN
         2012-06-17    2.532694
         2012-06-18    2.532694
         2012-06-21   -0.288329
         2012-06-22   -0.288329
         2012-06-29    1.191570
         dtype: float64
```

```
In [18]: ts2 + ts1.reindex(ts2.index, method='ffill')
```

```
Out[18]: 2012-06-12         NaN
         2012-06-17    3.127724
         2012-06-18    2.639646
         2012-06-21   -0.453418
         2012-06-22   -0.476010
         2012-06-29    1.504478
         dtype: float64
```

Using periods instead of timestamps Periods, as opposed to timestamps, are another common approach to organizing time series data, which can lead to its own time series frequency problems. Suppose you have the following GDP and inflation data, which are both periodic but at different frequencies:

```
In [19]: gdp = Series([1.78, 1.94, 2.08, 2.01, 2.15, 2.31, 2.46],
                      index=pd.period_range('1984Q2', periods=7, freq='Q-SEP'))
         infl = Series([0.025, 0.045, 0.037, 0.04],
                       index=pd.period_range('1982', periods=4, freq='A-DEC'))
         gdp
```

```
Out[19]: 1984Q2    1.78
         1984Q3    1.94
         1984Q4    2.08
         1985Q1    2.01
         1985Q2    2.15
         1985Q3    2.31
         1985Q4    2.46
         Freq: Q-SEP, dtype: float64
```

```
In [20]: infl
```

```
Out[20]: 1982    0.025
         1983    0.045
         1984    0.037
         1985    0.040
         Freq: A-DEC, dtype: float64
```

In `Pandas`, unlike with timestamps, operations between different-frequency time series that are indexed by periods are not possible without explicit conversions. In this case, if we know that `infl` values were observed at the end of the year, we can then convert to `Q-SEP` to get the right periods in that frequency:

```
In [21]: infl_q = infl.asfreq('Q-SEP', how='end')
         infl_q
```

```
Out[21]: 1983Q1    0.025
         1984Q1    0.045
         1985Q1    0.037
         1986Q1    0.040
         Freq: Q-SEP, dtype: float64
```

Then, we can simply reindex the inflation time series data with a forward-filling method to match the GDP data.

```
In [22]: infl_q.reindex(gdp.index, method='ffill')
```

```
Out[22]: 1984Q2    0.045
         1984Q3    0.045
         1984Q4    0.045
         1985Q1    0.037
         1985Q2    0.037
         1985Q3    0.037
         1985Q4    0.037
         Freq: Q-SEP, dtype: float64
```

1.1.3 Time of day and "as of" data selection

Suppose you have a long time series containing intraday market data and you want to extract the prices at a particular time of day on each day of the data. What if the data are irregular such that observations do not fall exactly on the desired time? In practice this task can make for error-prone data munging if you are not careful. Here is an example for illustration purposes:

```
In [23]: # Make an intraday date range and time series
         rng = pd.date_range('2012-06-01 09:30', '2012-06-01 15:59', freq='T')
         # Make a 5-day series of 9:30-15:59 values
         rng = rng.append([rng + pd.offsets.BDay(i) for i in range(1, 4)])
         ts = Series(np.arange(len(rng), dtype=float), index=rng)
         ts
```

```
Out[23]: 2012-06-01 09:30:00       0
         2012-06-01 09:31:00       1
         2012-06-01 09:32:00       2
         2012-06-01 09:33:00       3
         2012-06-01 09:34:00       4
         2012-06-01 09:35:00       5
                                 ...
         2012-06-06 15:54:00    1554
         2012-06-06 15:55:00    1555
         2012-06-06 15:56:00    1556
         2012-06-06 15:57:00    1557
         2012-06-06 15:58:00    1558
         2012-06-06 15:59:00    1559
         dtype: float64
```

We can index ts with a datetime.time object to extract the values at 10:00 AM throughout the entire time series.

```
In [24]: from datetime import time
         ts[time(10, 0)]
```

```
Out[24]: 2012-06-01 10:00:00      30
         2012-06-04 10:00:00     420
         2012-06-05 10:00:00     810
         2012-06-06 10:00:00    1200
         dtype: float64
```

Alternatively, you can specify timestamps between two time intervals, such as 10:00 AM and 10:01 AM (inclusively).

```
In [25]: ts.between_time(time(10, 0), time(10, 1))
```

```
Out[25]: 2012-06-01 10:00:00      30
         2012-06-01 10:01:00      31
         2012-06-04 10:00:00     420
         2012-06-04 10:01:00     421
         2012-06-05 10:00:00     810
         2012-06-05 10:01:00     811
         2012-06-06 10:00:00    1200
         2012-06-06 10:01:00    1201
         dtype: float64
```

```
In [26]: np.random.seed(12346)
```

However, it could be the case that no data actually fell exactly at 10:00 AM, but you might want to know the last known value at 10:00. In that case, you might do something like the following:

```
In [27]: # Set most of the time series randomly to NA
         indexer = np.sort(np.random.permutation(len(ts))[700:])
         irr_ts = ts.copy()
         irr_ts[indexer] = np.nan
         irr_ts['2012-06-01 09:50':'2012-06-01 10:00']
```

```
Out[27]: 2012-06-01 09:50:00    20
         2012-06-01 09:51:00   NaN
         2012-06-01 09:52:00    22
         2012-06-01 09:53:00    23
         2012-06-01 09:54:00   NaN
         2012-06-01 09:55:00    25
         2012-06-01 09:56:00   NaN
         2012-06-01 09:57:00   NaN
         2012-06-01 09:58:00   NaN
         2012-06-01 09:59:00   NaN
         2012-06-01 10:00:00   NaN
         dtype: float64
```

```
In [28]: selection = pd.date_range('2012-06-01 10:00', periods=4, freq='B')
         irr_ts.asof(selection)
```

```
Out[28]: 2012-06-01 10:00:00      25
         2012-06-04 10:00:00     420
         2012-06-05 10:00:00     810
         2012-06-06 10:00:00    1197
         Freq: B, dtype: float64
```

The `asof` method performs this functionality for you.

1.1.4 Splicing together data sources

In financial and economic contexts, there are a number of widely occurring use cases of merging related datasets:

- Switching from one data source to another at a specific point in time
- "Patching" missing values in a time series at the beginning, middle, or end using another time series
- Completely replacing the data for a subset of symbols

In the first case, switching from one set of time series to another at a specific instant, it is a matter of splicing together two `TimeSeries` or `DataFrame` objects using `pandas.concat`:

```
In [29]: data1 = DataFrame(np.ones((6, 3), dtype=float),
                           columns=['a', 'b', 'c'],
                           index=pd.date_range('6/12/2012', periods=6))
         data2 = DataFrame(np.ones((6, 3), dtype=float) * 2,
                           columns=['a', 'b', 'c'],
                           index=pd.date_range('6/13/2012', periods=6))
         spliced = pd.concat([data1.ix[:'2012-06-14'], data2.ix['2012-06-15':]])
         spliced
```

```
Out[29]:             a  b  c
         2012-06-12  1  1  1
         2012-06-13  1  1  1
         2012-06-14  1  1  1
         2012-06-15  2  2  2
         2012-06-16  2  2  2
         2012-06-17  2  2  2
         2012-06-18  2  2  2
```

Suppose in a similar example that `data1` was missing a time series present in `data2`:

```
In [30]: data2 = DataFrame(np.ones((6, 4), dtype=float) * 2,
                           columns=['a', 'b', 'c', 'd'],
                           index=pd.date_range('6/13/2012', periods=6))
         spliced = pd.concat([data1.ix[:'2012-06-14'], data2.ix['2012-06-15':]])
         spliced
```

```
Out[30]:             a  b  c   d
         2012-06-12  1  1  1 NaN
         2012-06-13  1  1  1 NaN
         2012-06-14  1  1  1 NaN
         2012-06-15  2  2  2   2
         2012-06-16  2  2  2   2
         2012-06-17  2  2  2   2
         2012-06-18  2  2  2   2
```

Using `combine_first`, you can bring in data from before the splice point to extend the history for `d` item:

```
In [31]: spliced_filled = spliced.combine_first(data2)
         spliced_filled
```

```
Out[31]:             a  b  c   d
         2012-06-12  1  1  1 NaN
         2012-06-13  1  1  1   2
         2012-06-14  1  1  1   2
         2012-06-15  2  2  2   2
```

```
2012-06-16  2  2  2   2
2012-06-17  2  2  2   2
2012-06-18  2  2  2   2
```

Since there isn't any data for `d` at 2012-06-12 in `data2`, the entry receives an `NaN`.

The `update` method for `DataFrame` objects performs in-place updates to the data. To fill in only holes in the data, pass in `overwrite=False`.

```
In [32]: spliced.update(data2, overwrite=False)

In [33]: spliced

Out[33]:             a  b  c   d
         2012-06-12  1  1  1 NaN
         2012-06-13  1  1  1   2
         2012-06-14  1  1  1   2
         2012-06-15  2  2  2   2
         2012-06-16  2  2  2   2
         2012-06-17  2  2  2   2
         2012-06-18  2  2  2   2
```

You can also perform data replacement on any subset of symbols. Below, this is used to fill in values for specific columns:

```
In [34]: cp_spliced = spliced.copy()
         cp_spliced[['a', 'c']] = data1[['a', 'c']]
         cp_spliced

Out[34]:              a  b    c   d
         2012-06-12   1  1    1 NaN
         2012-06-13   1  1    1   2
         2012-06-14   1  1    1   2
         2012-06-15   1  2    1   2
         2012-06-16   1  2    1   2
         2012-06-17   1  2    1   2
         2012-06-18 NaN  2  NaN   2
```

1.1.5 Return indexes and cumulative returns

The return of a financial asset is defined as the cumulative percent change in its price.

Here is some price data for Apple, Inc.:

```
In [35]: import pandas.io.data as web
         price = web.get_data_yahoo('AAPL', '2011-01-01')['Adj Close']
         price[-5:]

Out[35]: Date
         2015-07-13    125.660004
         2015-07-14    125.610001
         2015-07-15    126.820000
         2015-07-16    128.509995
         2015-07-17    129.619995
         Name: Adj Close, dtype: float64
```

Apple does not pay dividends, which would otherwise be included into the calculation of net returns. Thus, a quick-and-dirty computation of returns will suffice:

```
In [36]: price['2011-10-03'] / price['2011-3-01'] - 1
```

```
Out[36]: 0.072399969339113746
```

Stocks with dividend payments complicate the computation because you have to factor in the stream of payments over time. The `Adj Close` attempts to adjust for stock splits and dividends, but in any case, it's quite common to derive a *return index*, which indicates the value of a dollar's investment into the asset. For Apple, let's compute a simple return index using `cumprod`.

```
In [37]: returns = price.pct_change()
         ret_index = (1 + returns).cumprod()
         ret_index[0] = 1  # Set first value to 1
         ret_index.plot(grid=True)
```

```
Out[37]: <matplotlib.axes._subplots.AxesSubplot at 0x7f113c6fa050>
```

With a return index, we can manipulate the frequency at which we compute the returns quite easily.

```
In [38]: m_returns = ret_index.resample('BM', how='last').pct_change()
         m_returns['2012']
```

```
Out[38]: Date
         2012-01-31    0.127111
         2012-02-29    0.188311
         2012-03-30    0.105284
         2012-04-30   -0.025970
         2012-05-31   -0.010702
         2012-06-29    0.010853
         2012-07-31    0.045822
         2012-08-31    0.093877
```

```
2012-09-28    0.002796
2012-10-31   -0.107600
2012-11-30   -0.012374
2012-12-31   -0.090743
Freq: BM, Name: Adj Close, dtype: float64
```

Since no dividends or other adjustments are considered, we could have alternatively computed from the daily percent changed by resampling with a simple aggregation:

```
In [39]: m_rets = (1 + returns).resample('M', how='prod', kind='period') - 1
         m_rets['2012']
```

```
Out[39]: Date
         2012-01    0.127111
         2012-02    0.188311
         2012-03    0.105284
         2012-04   -0.025970
         2012-05   -0.010702
         2012-06    0.010853
         2012-07    0.045822
         2012-08    0.093877
         2012-09    0.002796
         2012-10   -0.107600
         2012-11   -0.012374
         2012-12   -0.090743
         Freq: M, Name: Adj Close, dtype: float64
```

Then, to include the dividend payments you can simply add the separate dividend payment data as follows:

```
returns[dividend_dates] += dividend_pcts
```

1.2 Group transforms and analysis

Let's consider a collection of made-up assets. We first generate a universe of 1000 tickers:

```
In [40]: pd.options.display.max_rows = 100
         pd.options.display.max_columns = 10
         np.random.seed(12345)
```

```
In [41]: import random; random.seed(0)
         import string

         N = 1000
         def rands(n):
             choices = string.ascii_uppercase
             return ''.join([random.choice(choices) for _ in xrange(n)])
         tickers = np.array([rands(5) for _ in xrange(N)])
```

Then, we can create a `DataFrame` containing three columns representing random portfolios for a given subset of the above tickers:

```
In [42]: M = 500
         df = DataFrame({'Momentum' : np.random.randn(M) / 200 + 0.03,
                         'Value' : np.random.randn(M) / 200 + 0.08,
                         'ShortInterest' : np.random.randn(M) / 200 - 0.02},
                         index=tickers[:M])
         df.head()
```

```
Out[42]:        Momentum  ShortInterest     Value
         VTKGN  0.028976      -0.024918  0.076191
         KUHMP  0.032395      -0.015345  0.078342
         XNHTQ  0.027403      -0.024058  0.071243
         GXZVX  0.027221      -0.029151  0.083144
         ISXRM  0.039829      -0.020694  0.081413
```

We can aggregate these random tickers by industry. In this simple example, let's use two industries: financial and technology. We can store the mapping as a `Series` object.

```
In [43]: ind_names = np.array(['FINANCIAL', 'TECH'])
         sampler = np.random.randint(0, len(ind_names), N)
         industries = Series(ind_names[sampler], index=tickers,
                             name='industry')
```

Using groupby mechanics, we can group `industries` and carry out group aggregation and transformations:

```
In [44]: by_industry = df.groupby(industries)
         by_industry.mean()

Out[44]:            Momentum  ShortInterest     Value
         industry
         FINANCIAL  0.029485      -0.020739  0.079929
         TECH       0.030407      -0.019609  0.080113
```

Of course, remember the handy `describe` method:

```
In [45]: by_industry.describe()

Out[45]:                    Momentum  ShortInterest       Value
         industry
         FINANCIAL count  246.000000     246.000000  246.000000
                   mean     0.029485      -0.020739    0.079929
                   std      0.004802       0.004986    0.004548
                   min      0.017210      -0.036997    0.067025
                   25%      0.026263      -0.024138    0.076638
                   50%      0.029261      -0.020833    0.079804
                   75%      0.032806      -0.017345    0.082718
                   max      0.045884      -0.006322    0.093334
         TECH      count  254.000000     254.000000  254.000000
                   mean     0.030407      -0.019609    0.080113
                   std      0.005303       0.005074    0.004886
                   min      0.016778      -0.032682    0.065253
                   25%      0.026456      -0.022779    0.076737
                   50%      0.030650      -0.019829    0.080296
                   75%      0.033602      -0.016923    0.083353
                   max      0.049638      -0.003698    0.093081
```

We can transform these portfolios along a particular industry by defining customized transformation functions. For example, standardizing within industry is widely used in equity portfolio research:

```
In [46]: # Within-Industry Standardize
         def zscore(group):
             return (group - group.mean()) / group.std()

         df_stand = by_industry.apply(zscore)
```

You can verify that each industry has mean (very nearly) 0 and standard deviation 1:

```
In [47]: df_stand.groupby(industries).agg(['mean', 'std'])
```

```
Out[47]:                 Momentum      ShortInterest              Value
                             mean std           mean std           mean std
         industry
         FINANCIAL  1.114736e-15   1  3.081772e-15   1  8.001278e-15   1
         TECH      -2.779929e-16   1 -1.910982e-15   1 -7.139521e-15   1
```

Other, built-in kinds of transforms, such as `rank`, can be used to make the analysis more concise.

```
In [48]: # Within-industry rank descending
         ind_rank = by_industry.rank(ascending=False)
         ind_rank.groupby(industries).agg(['min', 'max'])
```

```
Out[48]:           Momentum      ShortInterest      Value
                        min  max           min  max   min  max
         industry
         FINANCIAL        1  246             1  246     1  246
         TECH             1  254             1  254     1  254
```

In quantitative equity, "rank and standardize" is a common sequence of transformations. You can compose these operations concisely with a well-placed lambda, as follows:

```
In [49]: # Industry rank and standardize
         by_industry.apply(lambda x: zscore(x.rank())).head()
```

```
Out[49]:        Momentum  ShortInterest     Value
         VTKGN -0.091346      -0.976696 -1.004802
         KUHMP  0.794005       1.299919 -0.358356
         XNHTQ -0.541047      -0.836164 -1.679355
         GXZVX -0.583207      -1.623142  0.990749
         ISXRM  1.572120      -0.265423  0.374314
```

1.2.1 Group factor exposures

Quantitative portfolio management takes heavy advantage of *factor analysis*. From [Wikipedia][1]:

> Factor analysis is a statistical method used to describe variability among observed, correlated variables in terms of a potentially lower number of unobserved variables called factors. For example, it is possible that variations in four observed variables mainly reflect the variations in two unobserved variables. Factor analysis searches for such joint variations in response to unobserved latent variables.

Portfolio holdings and performance are decomposed using one or more factors represented as a portfolio of weights. A common example is a stock's *beta*, which measures co-movement between a stock and a benchmark (like the S&P 500). We can consider a contrived example of a portfolio constructed from three randomly-generated factors (usually called *factor loadings*) and some weights: [1]: https://en.wikipedia.org/wiki/Factor_analysis

```
In [50]: from numpy.random import rand
         fac1, fac2, fac3 = np.random.rand(3, 1000)

         ticker_subset = tickers.take(np.random.permutation(N)[:1000])

         # Weighted sum of factors plus noise
```

```
port = Series(0.7 * fac1 - 1.2 * fac2 + 0.3 * fac3 + rand(1000),
              index=ticker_subset)
factors = DataFrame({'f1': fac1, 'f2': fac2, 'f3': fac3},
                    index=ticker_subset)
```

Vector correlations between each factor and the portfolio may not indicate too much:

```
In [51]: factors.corrwith(port)
```

```
Out[51]: f1    0.402377
         f2   -0.680980
         f3    0.168083
         dtype: float64
```

The standard method to compute factor exposure is by least squares regression. You can do so with a number of Python libraries, from `SciPy` and `NumPy` to more advanced libraries such as `statsmodels`. However, `Pandas` makes the process particularly easy with its `pandas.ols` method.

```
In [52]: pd.ols(y=port, x=factors).beta
```

```
Out[52]: f1            0.761789
         f2           -1.208760
         f3            0.289865
         intercept     0.484477
         dtype: float64
```

Compare these with the original factor weights that were provided above arbitrarily, and you will see that this regression performed considerably better than the `corrwith` results; we have almost completely recovered the weights. With `groupby` you can compute exposures industry by industry. To do so, encapsulate the regression method with a new function, such as:

```
In [53]: def beta_exposure(chunk, factors=None):
             return pd.ols(y=chunk, x=factors).beta
```

Then, simply group by `industries` and apply the function, passing the `DataFrame` of factor loadings:

```
In [54]: by_ind = port.groupby(industries)
         exposures = by_ind.apply(beta_exposure, factors=factors)
         exposures.unstack()
```

```
Out[54]:                   f1        f2        f3  intercept
         industry
         FINANCIAL  0.790329 -1.182970  0.275624   0.455569
         TECH       0.740857 -1.232882  0.303811   0.508188
```

1.2.2 Decile and quartile analysis

In many circumstances it is useful to break down data based on sample quantiles. For example, the performance of a stock portfolio could be separated into quartiles based on each stock's price-to-earnings ratio. With `Pandas`, the method `pandas.qcut` combined with `groupby` makes this a straightforward task.

Consider a simple trend following or *momentum* strategy trading the S&P 500 index via the SPY exchange-traded fund.

```
In [55]: import pandas.io.data as web
         data = web.get_data_yahoo('SPY', '2006-01-01')
         data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 2401 entries, 2006-01-03 to 2015-07-17
Data columns (total 6 columns):
Open         2401 non-null float64
High         2401 non-null float64
Low          2401 non-null float64
Close        2401 non-null float64
Volume       2401 non-null int64
Adj Close    2401 non-null float64
dtypes: float64(5), int64(1)
memory usage: 131.3 KB
```

We compute the daily returns and a function for transforming the returns into a trend signal formed by a lagged moving sum:

```
In [56]: px = data['Adj Close']
         returns = px.pct_change()

         def to_index(rets):
             index = (1 + rets).cumprod()
             first_loc = max(index.index.get_loc(index.idxmax()) - 1, 0)
             index.values[first_loc] = 1
             return index

         def trend_signal(rets, lookback, lag):
             signal = pd.rolling_sum(rets, lookback, min_periods=lookback - 5)
             return signal.shift(lag)
```

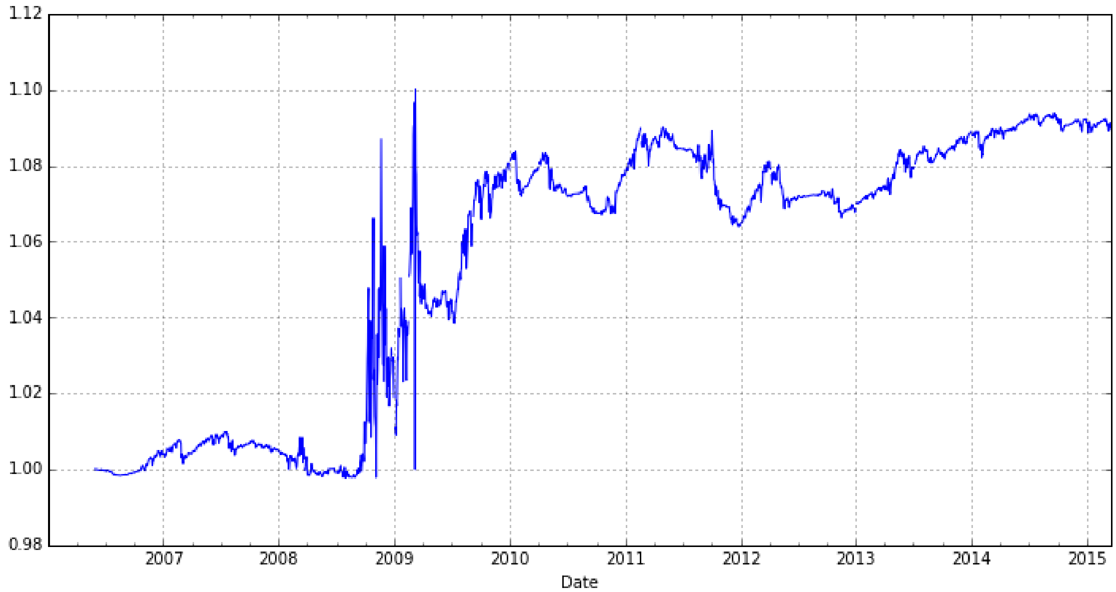
Using this function, we can create and test a trading strategy that trades this momentum signal every Friday:

```
In [57]: signal = trend_signal(returns, 100, 3)
         trade_friday = signal.resample('W-FRI').resample('B', fill_method='ffill')
         trade_rets = trade_friday.shift(1) * returns
         trade_rets = trade_rets[:len(returns)]
```

We can then convert the strategy returns to a return index and plot them:

```
In [58]: to_index(trade_rets).plot(grid=True, figsize=(12,6))

Out[58]: <matplotlib.axes._subplots.AxesSubplot at 0x7f110a4b6910>
```



Caveat: this is a naive strategy!

Suppose that you want to decompose the strategy performance into more and less volatile periods of trading. Trailing one-year annualized standard deviation is a simple measure of volatility, and we can compute Sharpe ratios to assess the reward-to-risk ratio in various volatility regimes:

```
In [59]: vol = pd.rolling_std(returns, 250, min_periods=200) * np.sqrt(250)

         def sharpe(rets, ann=250):
             return rets.mean() / rets.std()  * np.sqrt(ann)
```

Now we can divide `vol` into quartiles with `pd.qcut` and aggregating with `sharpe`:

```
In [60]: cats = pd.qcut(vol, 4)
         print('cats: %d, trade_rets: %d, vol: %d' % (len(cats), len(trade_rets), len(vol)))

cats: 2401, trade_rets: 2401, vol: 2401

In [61]: trade_rets.groupby(cats).agg(sharpe)

Out[61]: [0.0954, 0.115]    0.383027
         (0.115, 0.17]      0.261718
         (0.17, 0.217]     -0.016823
         (0.217, 0.457]     0.456884
         dtype: float64
```

These results show that the strategy performed the best during the period when the volatility was the highest.

1.3 More example applications

1.3.1 Future contract rolling

From *Python for Data Analysis*:

In practice, modeling and trading futures contracts on equities, currencies, commodities, bonds, and other asset classes is complicated by the time-limited nature of each contract. For example, at any given time for a type of future (say silver or copper futures) multiple contracts with different *expiration dates* may be traded. In many cases, the future contract expiring next (the *near* contract) will be the most liquid (highest volume and lowest bid-ask spread.

For the purposes of modeling and forecasting, it can be much easier to work with a *continuous* return index indicating the profit and loss associated with always holding the near contract. Transitioning from an expiring contract to the next (or *far*) contract is referred to as *rolling*. Computing a continuous future series from the individual contract data is not necessarily a straightforward exercise and typically requires a deeper understanding of the market and how the instruments are traded. For example, in practice when and how quickly would you trade out of an expiring contract and into the next contract?

We will go into one way to do so. Using scaled prices for the SPY exchange-traded fund as a proxy for the S&P 500, we have:

```
In [62]: pd.options.display.max_rows = 10
         import pandas.io.data as web
         # Approximate price of S&P 500 index
         px = web.get_data_yahoo('SPY')['Adj Close'] * 10
         px
```

```
Out[62]: Date
         2010-01-04    1014.45015
         2010-01-05    1017.13549
         2010-01-06    1017.85161
         2010-01-07    1022.14826
         2010-01-08    1025.54973
                          ...
         2015-07-13    2097.59995
         2015-07-14    2107.20001
         2015-07-15    2106.30005
         2015-07-16    2122.70004
         2015-07-17    2124.70001
         Name: Adj Close, dtype: float64
```

Now, a little bit of preamble. Let's put a couple of S&P 500 future contracts and expiry dates in a `Series` object.

```
In [63]: from datetime import datetime
         expiry = {'ESU2': datetime(2012, 9, 21),
                   'ESZ2': datetime(2012, 12, 21)}
         expiry = Series(expiry).order()
         expiry
```

```
Out[63]: ESU2   2012-09-21
         ESZ2   2012-12-21
         dtype: datetime64[ns]
```

Using Yahoo! Finance prices and a random walk with noise, we can simulate the two contracts into the future.

```
In [64]: np.random.seed(12347)
         N = 200
         walk = (np.random.randint(0, 200, size=N) - 100) * 0.25
```

```
perturb = (np.random.randint(0, 20, size=N) - 10) * 0.25
walk = walk.cumsum()

rng = pd.date_range(px.index[0], periods=len(px) + N, freq='B')
near = np.concatenate([px.values, px.values[-1] + walk])
far = np.concatenate([px.values, px.values[-1] + walk + perturb])
prices = DataFrame({'ESU2': near, 'ESZ2': far}, index=rng)
```

`prices` then has two time series for each contract that differ by a random amount.

```
In [65]: prices.tail()
```

```
Out[65]:                     ESU2        ESZ2
         2016-02-05  2153.95001  2155.70001
         2016-02-08  2140.20001  2142.45001
         2016-02-09  2148.20001  2149.95001
         2016-02-10  2164.70001  2163.95001
         2016-02-11  2144.70001  2142.45001
```

The technique that we will use to splice these two separate time series into a single continuous series is by constructing a weighting matrix. Active constraints have a weight of 1 until the expiry date gets close. At that point, we decide on a roll convention. Here is a function that computes a weighting matrix with linear decay:

```
In [66]: def get_roll_weights(start, expiry, items, roll_periods=5):
             # start : first date to compute weighting DataFrame
             # expiry : Series of ticker -> expiration dates
             # items : sequence of contract names

             dates = pd.date_range(start, expiry[-1], freq='B')
             weights = DataFrame(np.zeros((len(dates), len(items))),
                                 index=dates, columns=items)

             prev_date = weights.index[0]
             for i, (item, ex_date) in enumerate(expiry.iteritems()):
                 if i < len(expiry) - 1:
                     weights.ix[prev_date:ex_date - pd.offsets.BDay(), item] = 1
                     roll_rng = pd.date_range(end=ex_date - pd.offsets.BDay(),
                                              periods=roll_periods + 1, freq='B')

                     decay_weights = np.linspace(0, 1, roll_periods + 1)
                     weights.ix[roll_rng, item] = 1 - decay_weights
                     weights.ix[roll_rng, expiry.index[i + 1]] = decay_weights
                 else:
                     weights.ix[prev_date:, item] = 1

                 prev_date = ex_date

             return weights
```

The weights look like this around the ESU2 expiry:

```
In [67]: weights = get_roll_weights('6/1/2012', expiry, prices.columns)
         weights.ix['2012-09-12':'2012-09-21']
```

```
Out[67]:             ESU2  ESZ2
         2012-09-12   1.0   0.0
         2012-09-13   1.0   0.0
         2012-09-14   0.8   0.2
         2012-09-17   0.6   0.4
         2012-09-18   0.4   0.6
         2012-09-19   0.2   0.8
         2012-09-20   0.0   1.0
         2012-09-21   0.0   1.0
```

Finally, the rolled future returns are just a weighted sum of the contract returns:

```
In [68]: rolled_returns = (prices.pct_change() * weights).sum(1)
```

1.3.2 Rolling correlation and linear regression

From *Python for Data Analysis*:

> Dynamic models play an important role in financial modeling as they can be used to simulate trading decisions over a historical period. Moving window and exponentially-weighted time series functions are an example of tools that are used for dynamic models.
>
> Correlation is one way to look at the co-movement between the changes in two asset time series. panda's `rolling_corr` function can be called with two return series to compute the moving window correlation.

First, let's load some stocks from Yahoo! Finance and compute the daily returns.

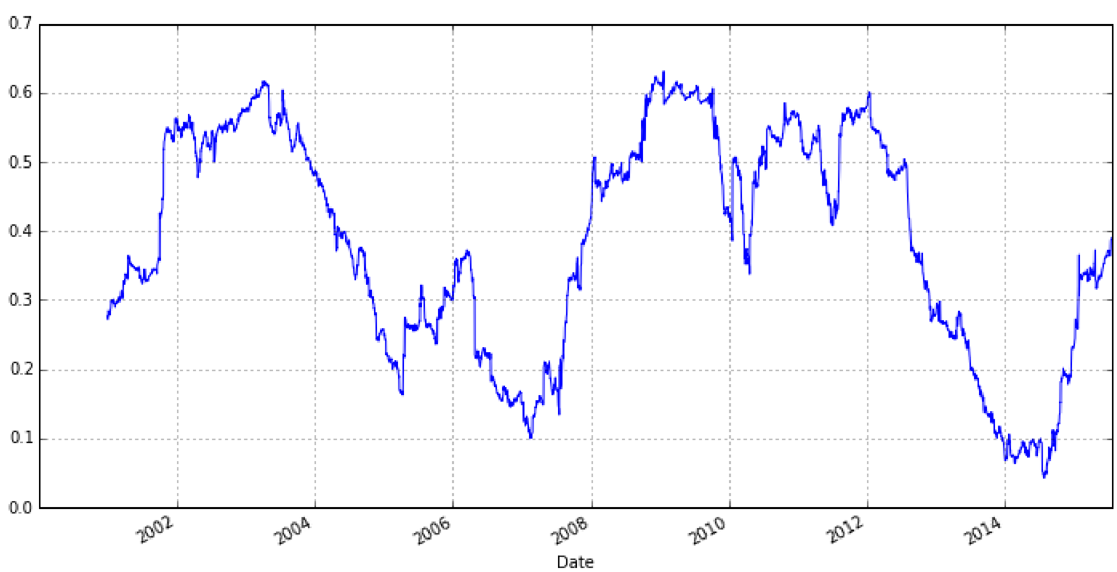
```
In [69]: aapl = web.get_data_yahoo('AAPL', '2000-01-01')['Adj Close']
         msft = web.get_data_yahoo('MSFT', '2000-01-01')['Adj Close']

         aapl_rets = aapl.pct_change()
         msft_rets = msft.pct_change()
```

Then, compute and plot the one-year moving correlation.

```
In [70]: plt.figure()
         pd.rolling_corr(aapl_rets, msft_rets, 250).plot(grid=True, figsize=(12,6))

Out[70]: <matplotlib.axes._subplots.AxesSubplot at 0x7f110a12ab90>
```



To better capture the differences in volatility, we can use least-squares regression. OLS regression can model the dynamic relationship between a variable and one or more other predictor variables.

```
In [71]: plt.figure()
         model = pd.ols(y=aapl_rets, x={'MSFT': msft_rets}, window=250)
         model.beta
```

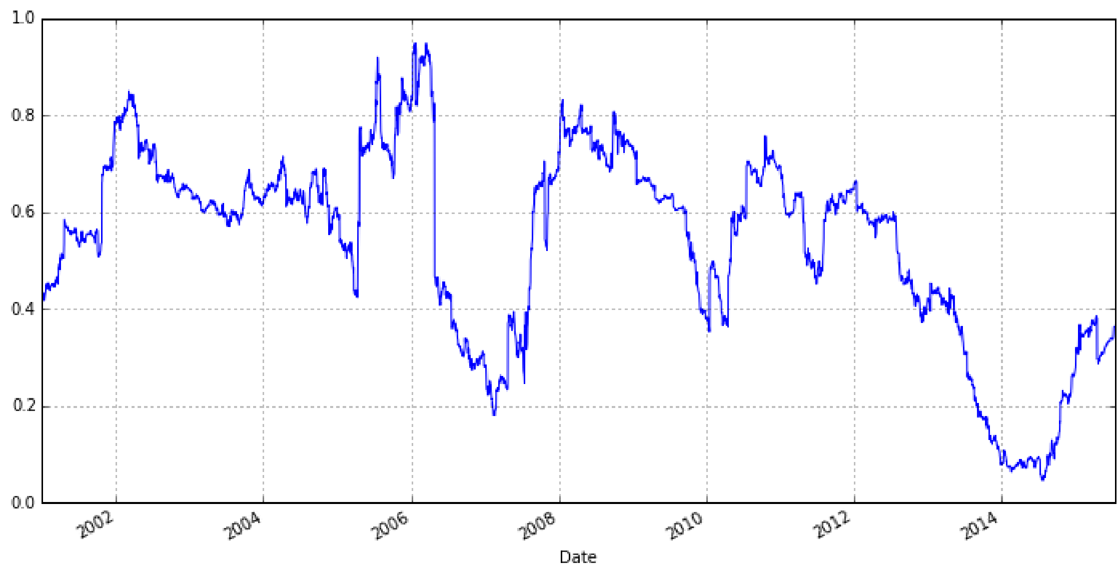
```
Out[71]:                  MSFT  intercept
         Date
         2000-12-28  0.429024  -0.002113
         2000-12-29  0.421105  -0.001796
         2001-01-02  0.420598  -0.001839
         2001-01-03  0.433294  -0.001289
         2001-01-04  0.432773  -0.001307
         ...              ...        ...
         2015-07-13  0.343578   0.001101
         2015-07-14  0.357289   0.001168
         2015-07-15  0.361721   0.001287
         2015-07-16  0.362751   0.001259
         2015-07-17  0.362994   0.001320

         [3659 rows x 2 columns]
```

```
<matplotlib.figure.Figure at 0x7f110a24c750>
```

```
In [72]: model.beta['MSFT'].plot(grid=True, figsize=(12,6))
```

```
Out[72]: <matplotlib.axes._subplots.AxesSubplot at 0x7f110a303c10>
```



There are of course more sophisticated techniques than OLS regression, which can be found in the statsmodels project, but this gives a flavor for the kind of analysis that is possible.