This repository Search Pull requests Issues Gist

quantopian / research_public

Watch 97 Star 181 Fork 106

Code Issues 0 Pull requests 0 Projects 0 Wiki Pulse Graphs

Branch: master research_public / lectures / Pairs Trading.ipynb Find file Copy path

mmargenot Changed rolling windows for beta and z-score d6697a5 15 days ago

1 contributor

1023 lines (1023 sloc) 818 KB Raw Blame History

Researching a Pairs Trading Strategy

By Delaney Granizo-Mackenzie and Maxwell Margenot

Part of the Quantopian Lecture Series:

- www.quantopian.com/lectures (https://www.quantopian.com/lectures)
- github.com/quantopian/research_public (https://github.com/quantopian/research_public)

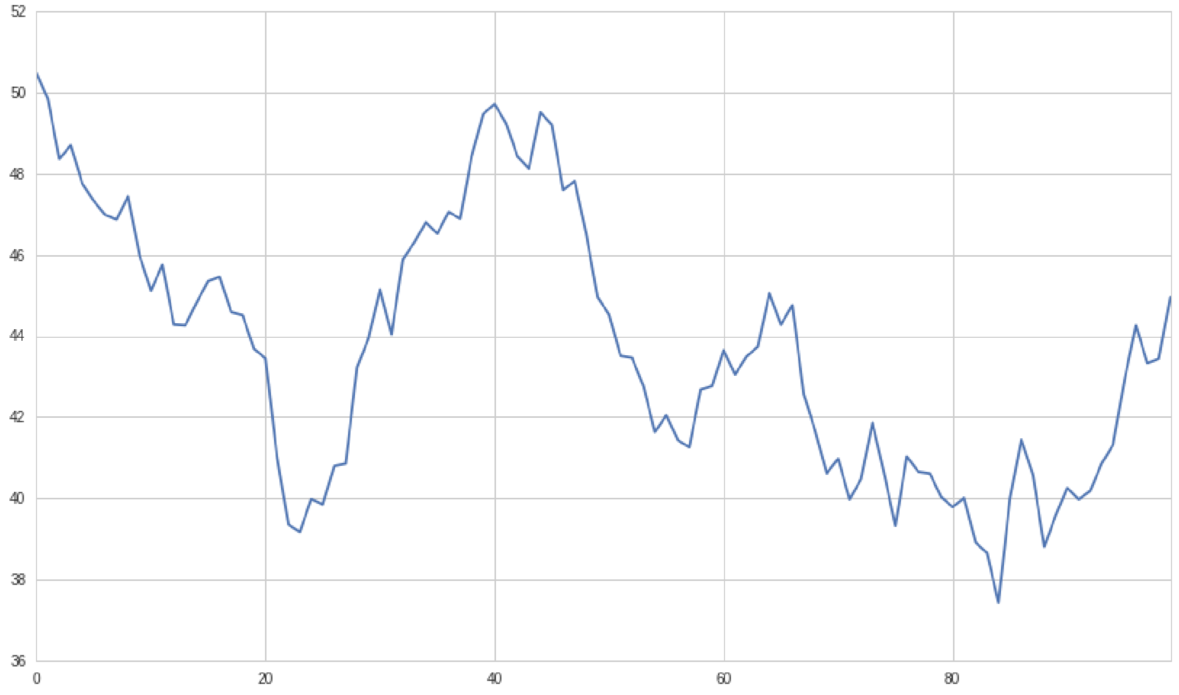
Pairs trading is a nice example of a strategy based on mathematical analysis. The principle is as follows. Let's say you have a pair of securities X and Y that have some underlying economic link. An example might be two companies that manufacture the same product, or two companies in one supply chain. We'll start by constructing an artificial example.

In [1]:
```
import numpy as np
import pandas as pd

import statsmodels
import statsmodels.api as sm
from statsmodels.tsa.stattools import coint
# just set the seed for the random number generator
np.random.seed(107)

import matplotlib.pyplot as plt
```

Explaining the Concept: We start by generating two fake securities.

We model X's daily returns by drawing from a normal distribution. Then we perform a cumulative sum to get the value of X on each day.

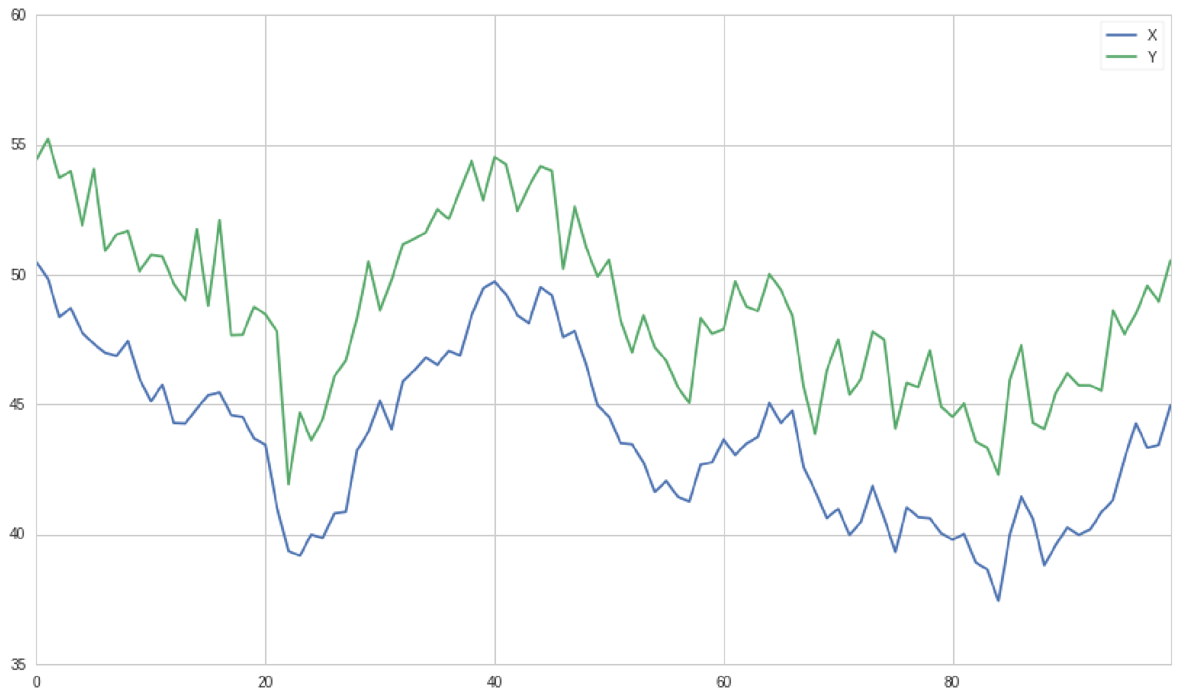
In [2]:
```
X_returns = np.random.normal(0, 1, 100) # Generate the daily returns
# sum them and shift all the prices up into a reasonable range
X = pd.Series(np.cumsum(X_returns), name='X') + 50
X.plot();
```



Now we generate Y. Remember that Y is supposed to have a deep economic link to X, so the price of Y should vary pretty similarly. We model this by taking X, shifting it up and adding some random noise drawn from a normal distribution.

In [3]:
```
some_noise = np.random.normal(0, 1, 100)
Y = X + 5 + some_noise
Y.name = 'Y'
```

```
pd.concat([X, Y], axis=1).plot();
```



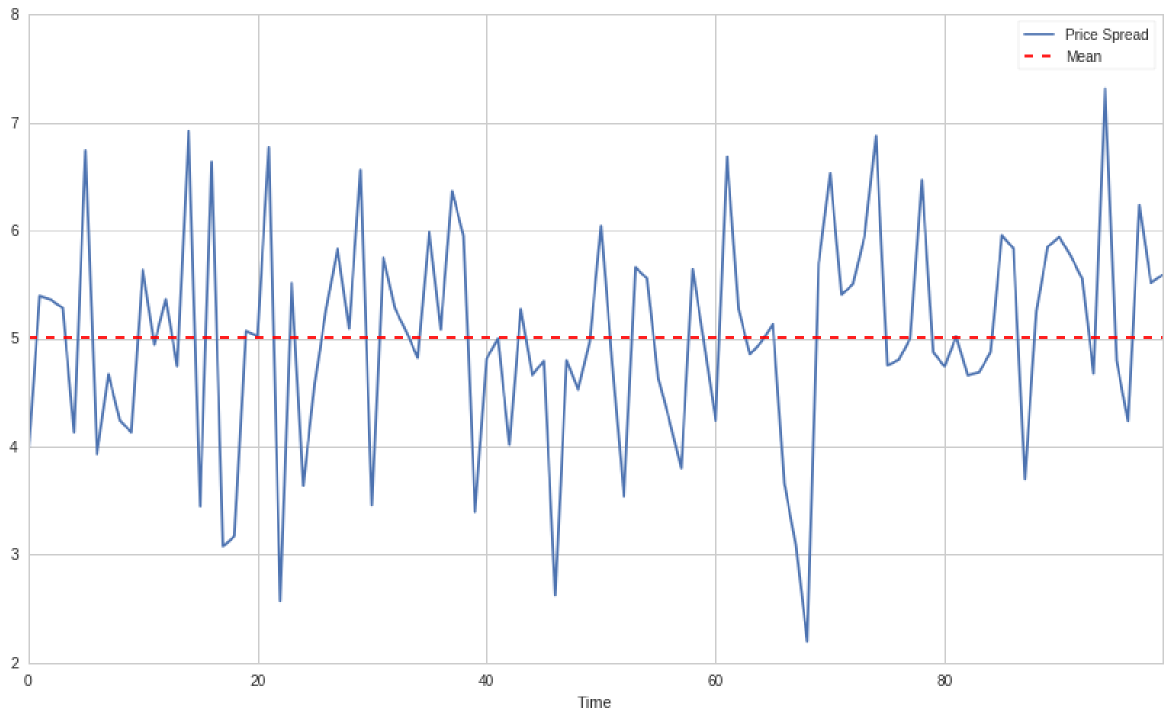
Def: Cointegration

We've constructed an example of two cointegrated series. Cointegration is a "different" form of correlation (very loosely speaking). If two timeseries are cointegrated, there is some linear combination between them that will vary around a mean. At all points in time, the combination between them is related to the same probability distrribution.

For more details on how we formally define cointegration and how to understand it, please see the Integration, Cointegration, and Stationarity Lecture from the Quantopian Lecture Series (https://www.quantopian.com/lectures#Integration,-Cointegration,-and-Stationarity).

We'll plot the difference between the two now so we can see how this looks.

In [4]:
```
(Y - X).plot() # Plot the spread
plt.axhline((Y - X).mean(), color='red', linestyle='--') # Add the mean
plt.xlabel('Time')
plt.legend(['Price Spread', 'Mean']);
```



Testing for Cointegration

That's an intuitive definition, but how do we test for this statisitcally? There is a convenient test that lives in `statsmodels.tsa.stattools`. We should see a very low p-value, as we've artifically created two series that are as cointegrated as physically possible.

In [5]:
```
# compute the p-value of the cointegration test
# will inform us as to whether the spread between the 2 timeseries is stationary
# around its mean
score, pvalue, _ = coint(X,Y)
print pvalue
```

```
2.75767345363e-16
```

Correlation vs. Cointegration

Correlation and cointegration, while theoretically similar, are not the same. To demonstrate this, we'll show examples of series that are correlated, but not cointegrated, and vice versa. To start let's check the correlation of the series we just generated.

In [6]:
```
X.corr(Y)
```

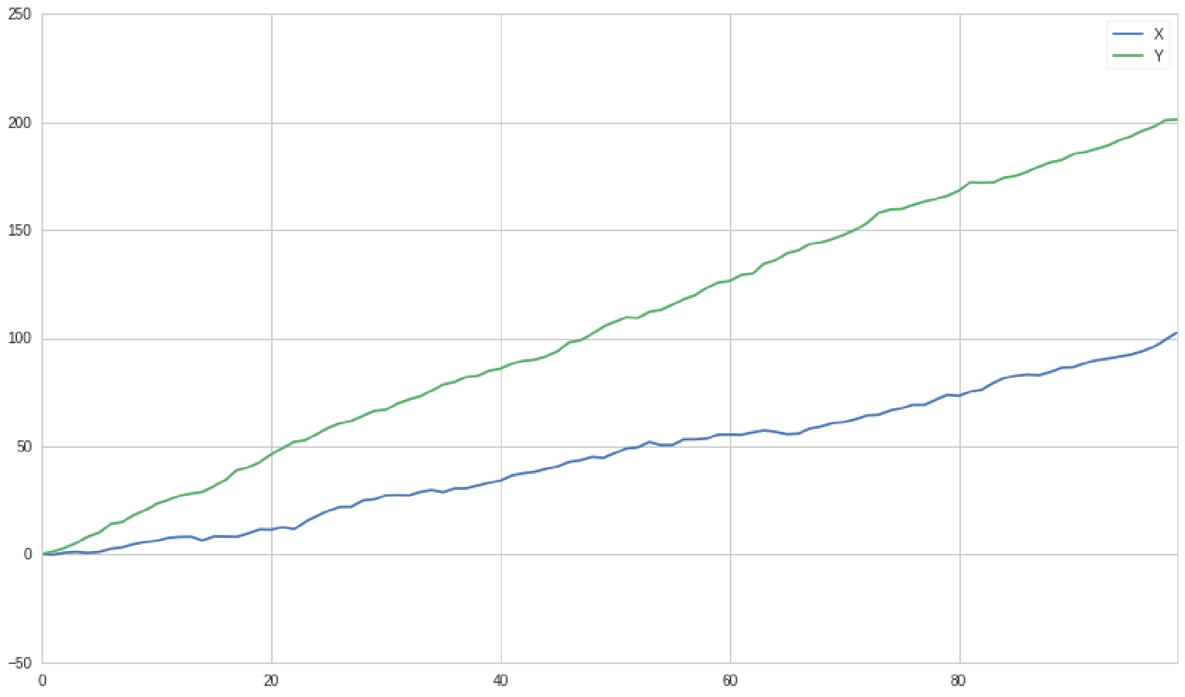
Out[6]:
```
0.94970906463859317
```

That's very high, as we would expect. But how would two series that are correlated but not cointegrated look?

Correlation Without Cointegration

A simple example is two series that just diverge.

In [7]:
```
X_returns = np.random.normal(1, 1, 100)
Y_returns = np.random.normal(2, 1, 100)

X_diverging = pd.Series(np.cumsum(X_returns), name='X')
Y_diverging = pd.Series(np.cumsum(Y_returns), name='Y')

pd.concat([X_diverging, Y_diverging], axis=1).plot();
```



In [8]:
```
print 'Correlation: ' + str(X_diverging.corr(Y_diverging))
score, pvalue, _ = coint(X_diverging,Y_diverging)
print 'Cointegration test p-value: ' + str(pvalue)
```

```
Correlation: 0.993134380128
Cointegration test p-value: 0.884633444839
```

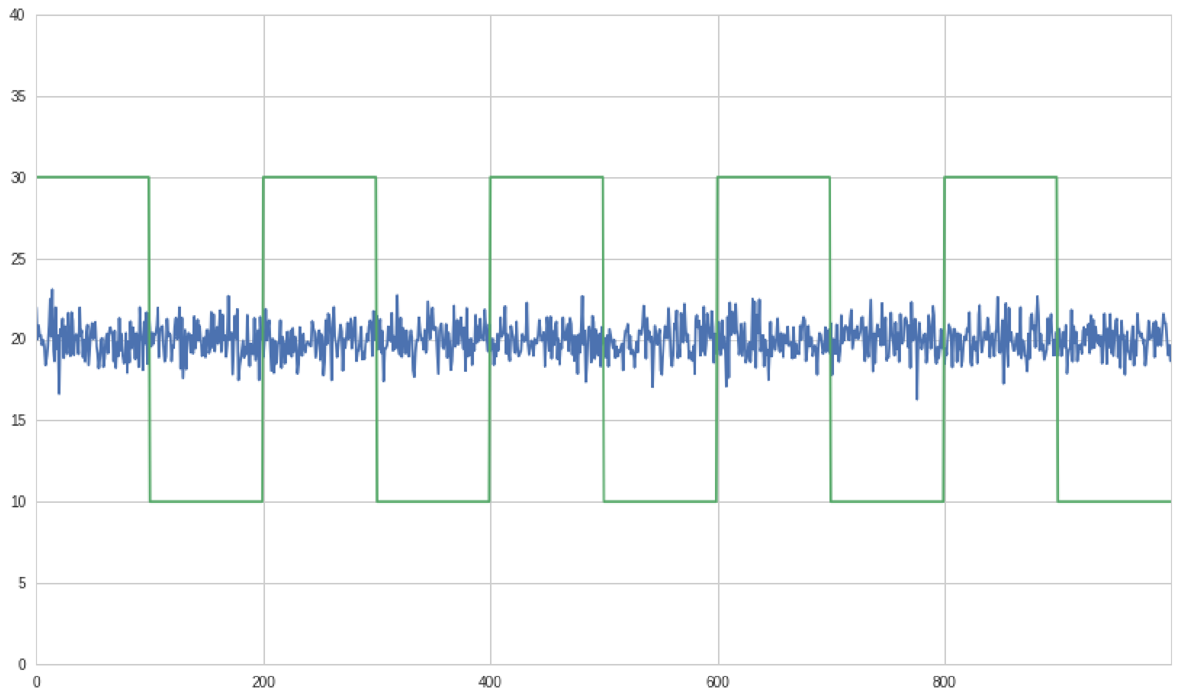
Cointegration Without Correlation

A simple example of this case is a normally distributed series and a square wave.

In [9]:
```
Y2 = pd.Series(np.random.normal(0, 1, 1000), name='Y2') + 20
Y3 = Y2.copy()
```

In [10]:
```
# Y2 = Y2 + 10
Y3[0:100] = 30
Y3[100:200] = 10
Y3[200:300] = 30
Y3[300:400] = 10
Y3[400:500] = 30
Y3[500:600] = 10
Y3[600:700] = 30
Y3[700:800] = 10
Y3[800:900] = 30
Y3[900:1000] = 10
```

In [11]:
```
Y2.plot()
Y3.plot()
plt.ylim([0, 40]);
```



In [12]:
```
# correlation is nearly zero
print 'Correlation: ' + str(Y2.corr(Y3))
score, pvalue, _ = coint(Y2,Y3)
print 'Cointegration test p-value: ' + str(pvalue)
```

```
Correlation: -0.0413040695809
Cointegration test p-value: 0.0
```

Sure enough, the correlation is incredibly low, but the p-value shows that these are clearly cointegrated.

Def: Hedged Position

Because you'd like to protect yourself from bad markets, often times short sales will be used to hedge long investments. Because a short sale makes money if the security sold loses value, and a long purchase will make money if a security gains value, one can long parts of the market and short others. That way if the entire market falls off a cliff, we'll still make money on the shorted securities and hopefully break even. In the case of two securities we'll call it a hedged position when we are long on one security and short on the other.

The Trick: Where it all comes together

Because the securities drift towards and apart from each other, there will be times when the distance is high and times when the distance is low. The trick of pairs trading comes from maintaining a hedged position across X and Y. If both securities go down, we neither make nor lose money, and likewise if both go up. We make money on the spread of the two reverting to the mean. In order to do this we'll watch for when X and Y are far apart, then short Y and long X. Similarly we'll watch for when they're close together, and long Y and short X.

Going Long the Spread

This is when the spread is small and we expect it to become larger. We place a bet on this by longing Y and shorting X.

Going Short the Spread

This is when the spread is large and we expect it to become smaller. We place a bet on this by shorting Y and longing X.

Specific Bets

One important concept here is that we are placing a bet on one specific thing, and trying to reduce our bet's dependency on other factors such as the market.

Finding real securities that behave like this

The best way to do this is to start with securities you suspect may be cointegrated and perform a statistical test. If you just run statistical tests over all pairs, you'll fall prey to multiple comparison bias.

Here's a method I wrote to look through a list of securities and test for cointegration between all pairs. It returns a cointegration test score matrix, a p-value matrix, and any pairs for which the p-value was less than 0.05.

WARNING: This will incur a large amount of multiple comparisons bias.

For those not familiar with multiple comparisons bias, it is the increased chance to incorrectly generate a significant p-value when many tests are run. If 100 tests are run on random data, we should expect to see 5 p-values below 0.05 on expectation. Because we will perform $n(n-1)/2$ comparisons, we should expect to see many incorrectly significant p-values. For the sake of example will will ignore this and continue. In practice a second verification step would be needed if looking for pairs this way. Another approach is to pick a small number of pairs you have reason to suspect might be cointegrated and test each individually. This will result in less exposure to multiple comparisons bias.

In [13]:
```
def find_cointegrated_pairs(data):
    n = data.shape[1]
    score_matrix = np.zeros((n, n))
    pvalue_matrix = np.ones((n, n))
    keys = data.keys()
    pairs = []
    for i in range(n):
        for j in range(i+1, n):
            S1 = data[keys[i]]
            S2 = data[keys[j]]
            result = coint(S1, S2)
            score = result[0]
            pvalue = result[1]
            score_matrix[i, j] = score
            pvalue_matrix[i, j] = pvalue
            if pvalue < 0.05:
                pairs.append((keys[i], keys[j]))
    return score_matrix, pvalue_matrix, pairs
```

Looking for Cointegrated Pairs of Alternative Energy Securities

I'm looking through a set of solar company stocks to see if any of them are cointegrated. We'll start by defining the list of securities we want to look through. Then we'll get the pricing data for each security for the year of 2014.

Note

We include the market in our data. This is because the market drives the movement of so many securities that you often times might find two seemingingly cointegrated securities, but in reality they are not cointegrated and just both conintegrated with the market. This is known as a confounding variable and it is important to check for market involvement in any relationship you find.

get_pricing() is a Quantopian method that pulls in stock data, and loads it into a Python Pandas DataPanel object. Available fields are 'price', 'open_price', 'high', 'low', 'volume'. But for this example we will just use 'price' which is the daily closing price of the stock.

In [14]:
```
symbol_list = ['ABGB', 'ASTI', 'CSUN', 'DQ', 'FSLR','SPY']
prices_df = get_pricing(symbol_list, fields=['price']
                               , start_date='2014-01-01', end_date='2015-01-01')['price']
prices_df.columns = map(lambda x: x.symbol, prices_df.columns)
```

Example of how to get all the prices of all the stocks loaded using get_pricing() above in one pandas dataframe object

```
In [15]: prices_df.head()
```

Out[15]:

	ABGB	ASTI	CSUN	DQ	FSLR	SPY
2014-01-02 00:00:00+00:00	14.4198	7.409	7.0400	38.00	57.43	182.95
2014-01-03 00:00:00+00:00	14.7546	7.250	7.0775	39.50	56.74	182.80
2014-01-06 00:00:00+00:00	15.3300	7.121	7.0000	40.05	51.26	182.40
2014-01-07 00:00:00+00:00	15.6300	7.299	6.9300	41.93	52.48	183.44
2014-01-08 00:00:00+00:00	15.3100	7.101	7.1600	42.49	51.68	183.53

Example of how to get just the prices of a single stock that was loaded using get_pricing() above

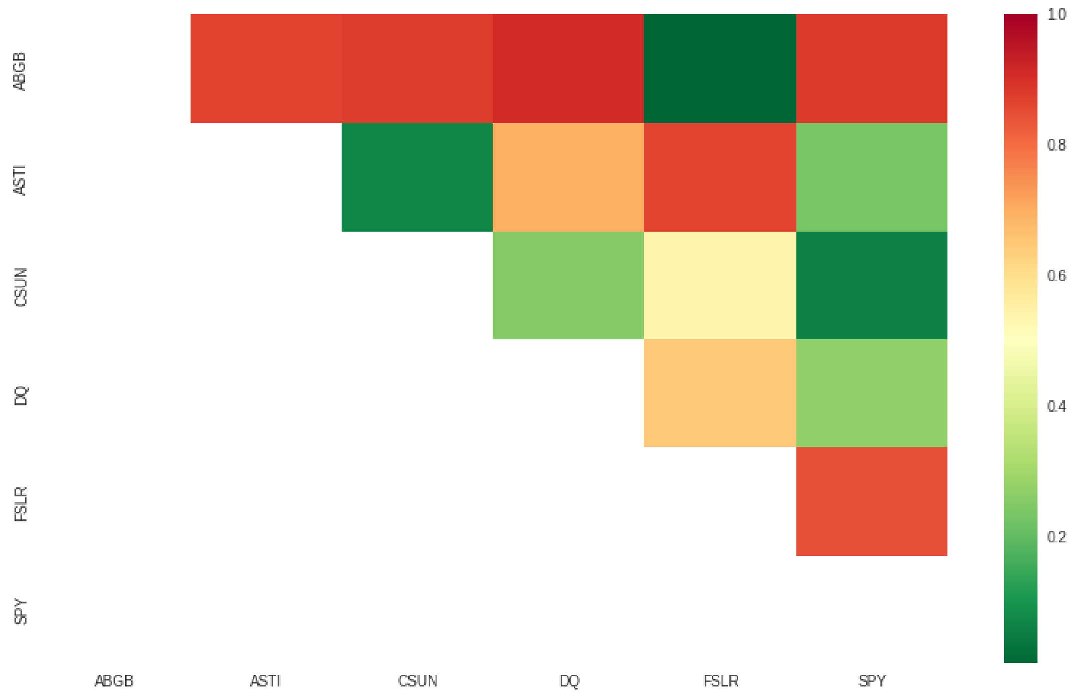
```
In [16]: prices_df['SPY'].head()
```

```
Out[16]: 2014-01-02 00:00:00+00:00    182.95
         2014-01-03 00:00:00+00:00    182.80
         2014-01-06 00:00:00+00:00    182.40
         2014-01-07 00:00:00+00:00    183.44
         2014-01-08 00:00:00+00:00    183.53
         Name: SPY, dtype: float64
```

Now we'll run our method on the list and see if any pairs are cointegrated.

```
In [17]: # Heatmap to show the p-values of the cointegration test between each pair of
         # stocks. Only show the value in the upper-diagonal of the heatmap

         scores, pvalues, pairs = find_cointegrated_pairs(prices_df)
         import seaborn
         seaborn.heatmap(pvalues, xticklabels=symbol_list, yticklabels=symbol_list, cmap='RdYlGn_r'
                         , mask = (pvalues >= 0.95)
                         )
         print pairs
```

```
[(u'ABGB', u'FSLR')]
```



Looks like 'ABGB' and 'FSLR' are cointegrated. Let's take a look at the prices to make sure there's nothing weird going on.

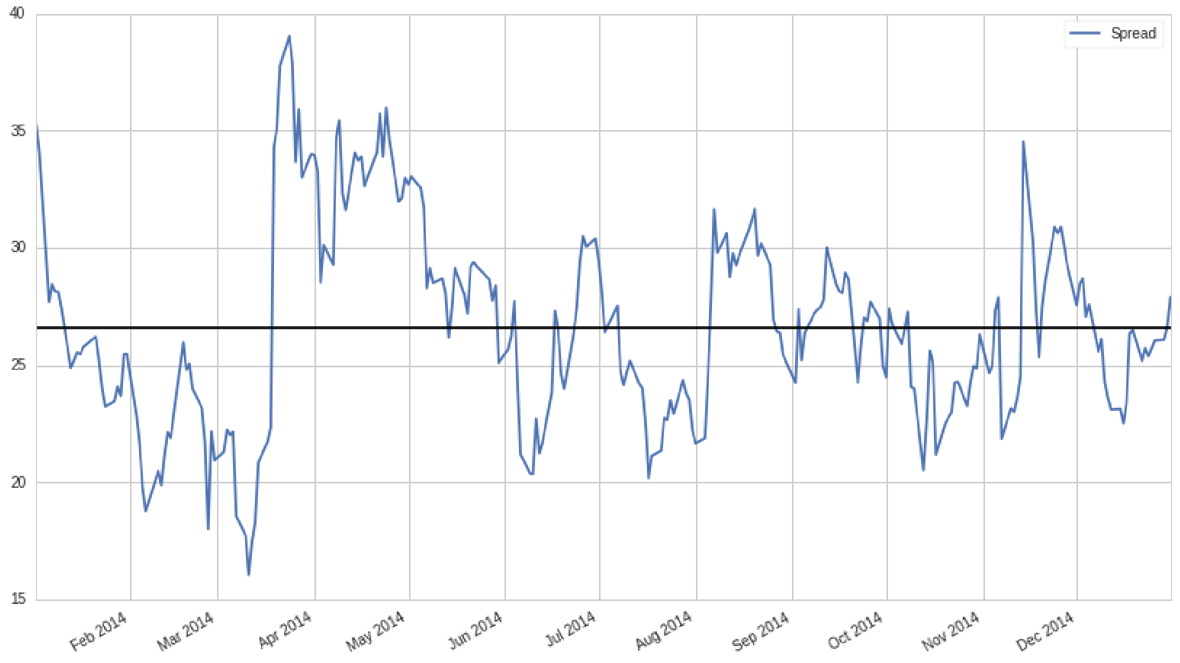
```
In [18]: S1 = prices_df['ABGB']
         S2 = prices_df['FSLR']
```

```
In [19]: score, pvalue, _ = coint(S1, S2)
         pvalue
```

```
Out[19]: 0.0060792512281641932
```

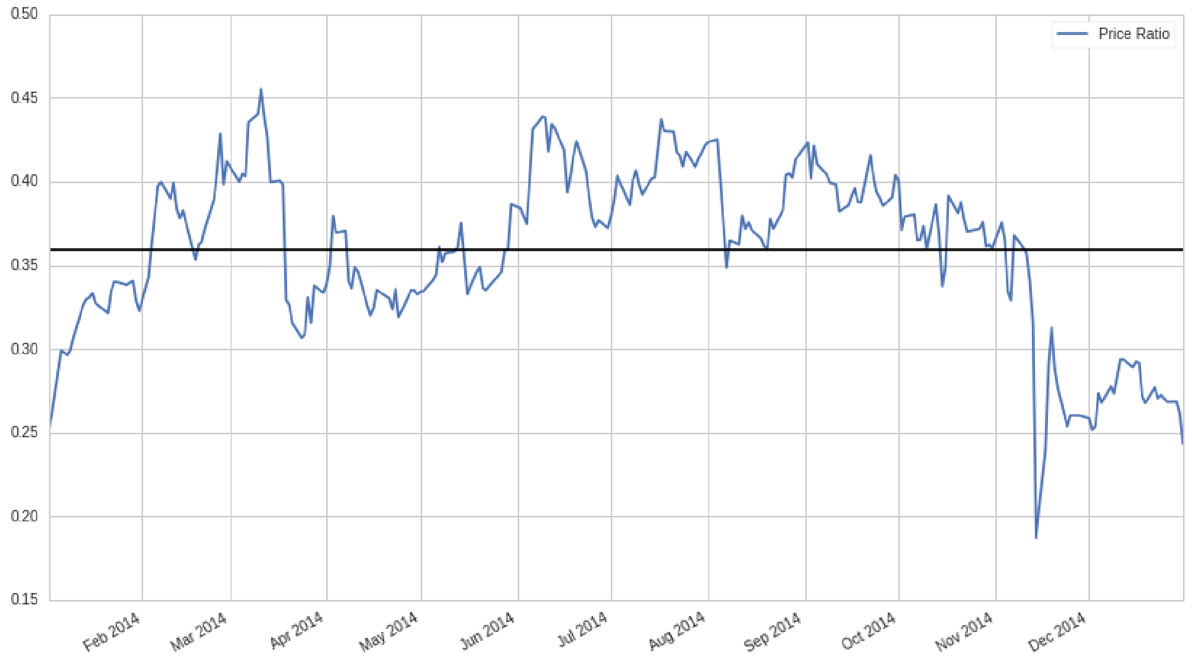
We'll plot the spread of the two series. Here we use linear regression to calculate a coefficient for their linear combination.

```
In [20]: S1 = sm.add_constant(S1)
         results = sm.OLS(S2, S1).fit()
         S1 = S1['ABGB']
         b = results.params['ABGB']

         spread = S2 - b * S1
         spread.plot()
         plt.axhline(spread.mean(), color='black')
         plt.legend(['Spread']);
```



Alternatively, we could examine the ratio betwen the two series.

```
In [21]: ratio = S1/S2
         ratio.plot()
         plt.axhline(ratio.mean(), color='black')
         plt.legend(['Price Ratio']);
```



Examining the price ratio of a trading pair is a traditional way to handle pairs trading. Part of why this works as a signal is based in our assumptions of how stock prices move, specifically because stock prices are typically assumed to be log-normally distributed. What this implies is that by taking a ratio of the prices, we are taking a linear combination of the returns associated with them (since prices are just the exponentiated returns).

This can be a little irritating to deal with for our purposes as purchasing the precisely correct ratio of a trading pair may not be practical. We choose instead to move forward with simply calculating the spread between the cointegrated stocks using linear regression. This is a very simple way to handle the relationship, however, and is likely not feasible for non-toy examples. There are other potential methods for estimating the spread listed at the bottom of this lecture. If you want to get more into the theory of why having cointegrated stocks matters for pairs trading, again, please see the Integration, Cointegration, and Stationarity Lecture from the Quantopian Lecture Series (https://www.quantopian.com/lectures#Integration,-Cointegration,-and-Stationarity).

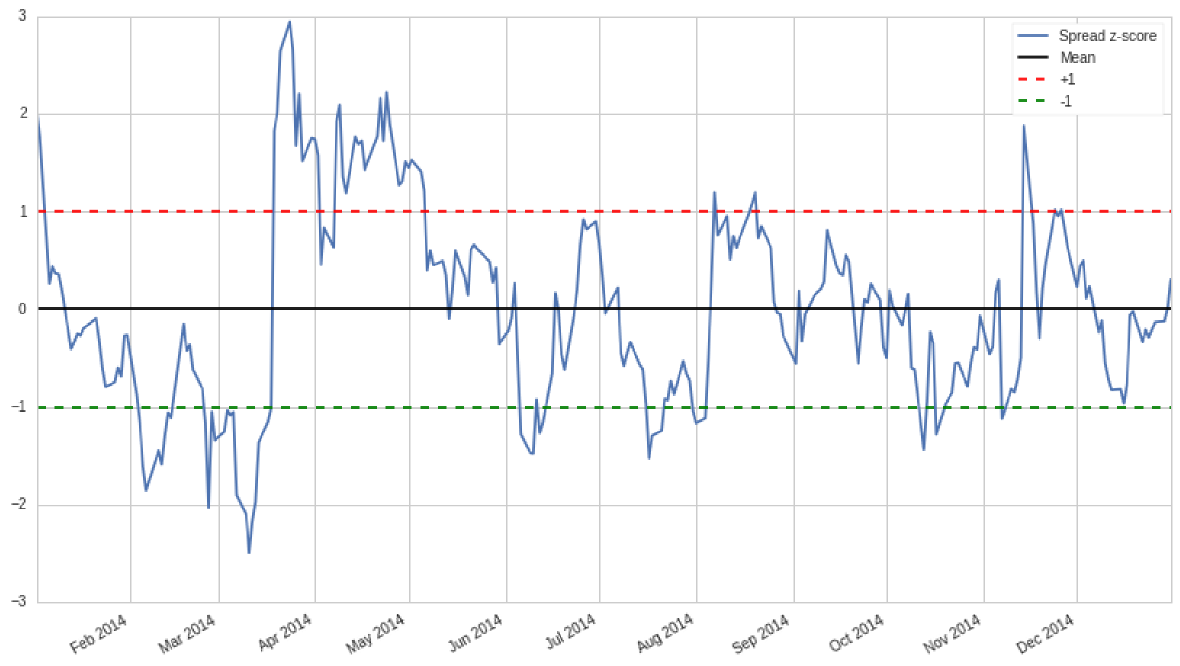
So, back to our example. The absolute spread isn't very useful in statistical terms. It is more helpful to normalize our signal by treating it as a z-score.

WARNING

In practice this is usually done to try to give some scale to the data, but this assumes some underlying distribution. Usually normal. Under a normal distribution, we would know that approximately 84% of all spread values will be smaller. However, much financial data is not normally distributed, and one must be very careful not to assume normality, nor any specific distribution when generating statistics. It could be the case that the true distribution of spreads was very fat-tailed and prone to extreme values. This could mess up our model and result in large losses.

```
In [22]: def zscore(series):
             return (series - series.mean()) / np.std(series)
```

```
In [23]: zscore(spread).plot()
         plt.axhline(zscore(spread).mean(), color='black')
         plt.axhline(1.0, color='red', linestyle='--')
         plt.axhline(-1.0, color='green', linestyle='--')
         plt.legend(['Spread z-score', 'Mean', '+1', '-1']);
```



Simple Strategy:

- Go "Long" the spread whenever the z-score is below -1.0
- Go "Short" the spread when the z-score is above 1.0
- Exit positions when the z-score approaches zero

This is just the tip of the iceberg, and only a very simplistic example to illustrate the concepts. In practice you would want to compute a more optimal weighting for how many shares to hold for S1 and S2. Some additional resources on pair trading are listed at the end of this notebook

Trading using constantly updating statistics

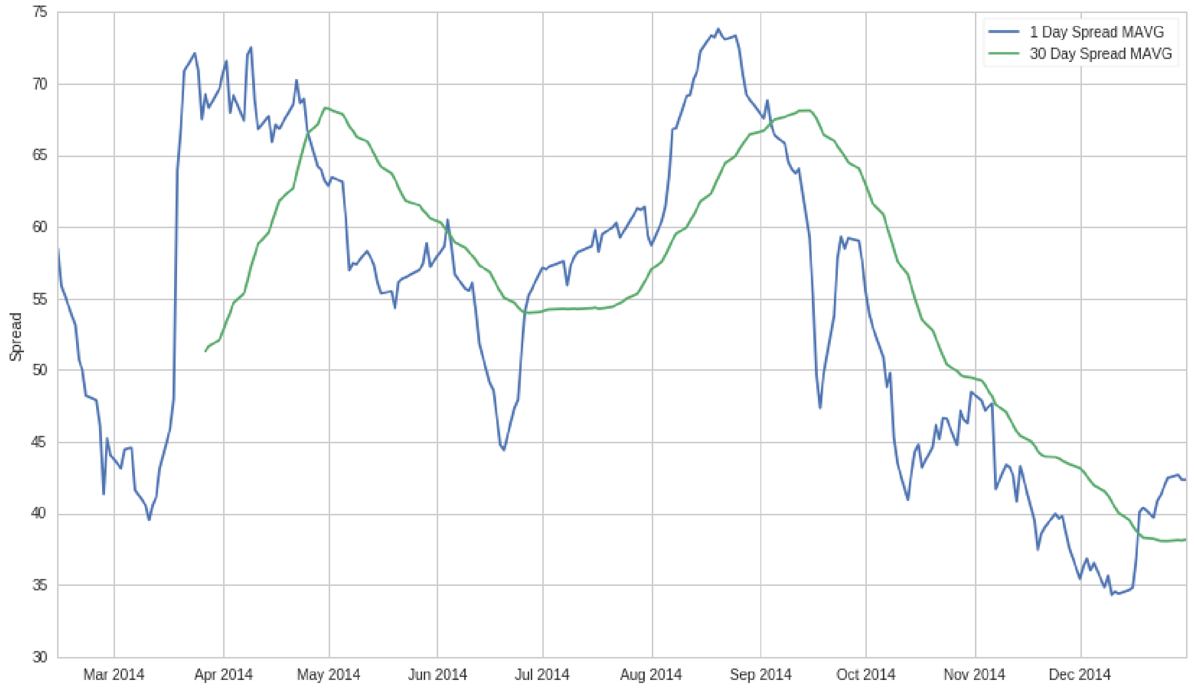
In general taking a statistic over your whole sample size can be bad. For example, if the market is moving up, and both securities with it, then your average price over the last 3 years may not be representative of today. For this reason traders often use statistics that rely on rolling windows of the most recent data.

Def: Moving Average

A moving average is just an average over the last n datapoints for each given time. It will be undefined for the first n datapoints in our series. Shorter moving averages will be more jumpy and less reliable, but respond to new information quickly. Longer moving averages will be smoother, but take more time to incorporate new information.

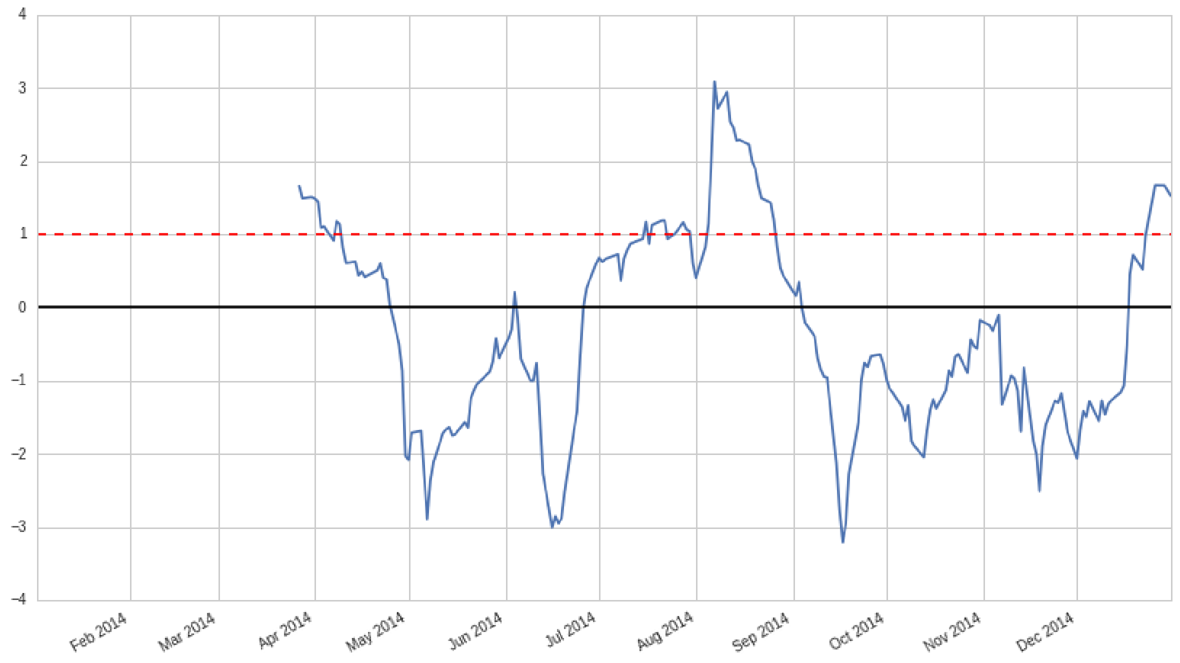
We also need to use a rolling beta, a rolling estimate of how our spread should be calculated, in order to keep all of our parameters up to date.

In [24]:
```
# Get the spread between the 2 stocks
# Calculate rolling beta coefficient
rolling_beta = pd.ols(y=S1, x=S2, window_type='rolling', window=30)
spread = S2 - rolling_beta.beta['x'] * S1
spread.name = 'spread'

# Get the 1 day moving average of the price spread
spread_mavg1 = pd.rolling_mean(spread, window=1)
spread_mavg1.name = 'spread 1d mavg'

# Get the 30 day moving average
spread_mavg30 = pd.rolling_mean(spread, window=30)
spread_mavg30.name = 'spread 30d mavg'

plt.plot(spread_mavg1.index, spread_mavg1.values)
plt.plot(spread_mavg30.index, spread_mavg30.values)

plt.legend(['1 Day Spread MAVG', '30 Day Spread MAVG'])

plt.ylabel('Spread');
```



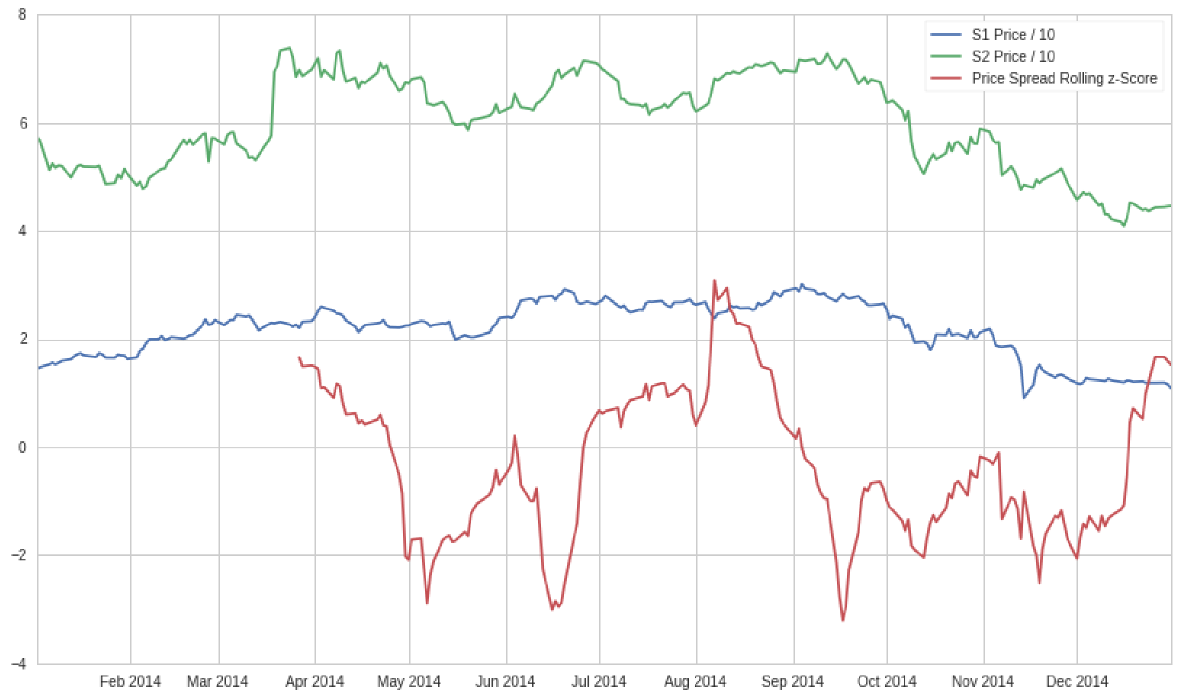
We can use the moving averages to compute the z-score of the spread at each given time. This will tell us how extreme the spread is and whether it's a good idea to enter a position at this time. Let's take a look at the z-score now.

In [25]:
```
# Take a rolling 30 day standard deviation
std_30 = pd.rolling_std(spread, window=30)
std_30.name = 'std 30d'

# Compute the z score for each day
zscore_30_1 = (spread_mavg1 - spread_mavg30)/std_30
zscore_30_1.name = 'z-score'
zscore_30_1.plot()
plt.axhline(0, color='black')
plt.axhline(1.0, color='red', linestyle='--');
```



The z-score doesn't mean much out of context, let's plot it next to the prices to get an idea of what it looks like. We'll take the negative of the z-score because the spreads were all negative and that is a little counterintuitive to trade on.

In [26]:
```
# Plot the prices scaled down along with the negative z-score
# just divide the stock prices by 10 to make viewing it on the plot easier
plt.plot(S1.index, S1.values/10)
plt.plot(S2.index, S2.values/10)
plt.plot(zscore_30_1.index, zscore_30_1.values)
plt.legend(['S1 Price / 10', 'S2 Price / 10', 'Price Spread Rolling z-Score']);
```



Implementation

When actually implementing a pairs trading strategy you would normally want to be trading many different pairs at once. If you find a good pair relationship by analyzing data, there is no guarantee that that relationship will continue into the future. Trading many different pairs creates a diversified portfolio to mitigate the risk of individual pairs "falling out of" cointegration.

There is a template algorithm attached to this lecture (https://www.quantopian.com/lectures#Introduction-to-Pairs-Trading) that shows an example of how you would implement pairs trading on our platform. Feel free to check it out and modify it with your own pairs to see if you can improve it.

This notebook contained some simple introductory approaches. In practice one should use more sophisticated statistics, some of which are listed here.

- Augmented-Dickey Fuller test
- Hurst exponent
- Half-life of mean reversion inferred from an Ornstein–Uhlenbeck process
- Kalman filters

(this is *not* an endorsement) But, a very good practical resource for learning more about pair trading is Dr. Ernie Chan's book: Algorithmic Trading: Winning Strategies and Their Rationale http://www.amazon.com/Algorithmic-Trading-Winning-Strategies-Rationale/dp/1118460146/ (http://www.amazon.com/Algorithmic-Trading-Winning-Strategies-Rationale/dp/1118460146/)