ThinkDSP

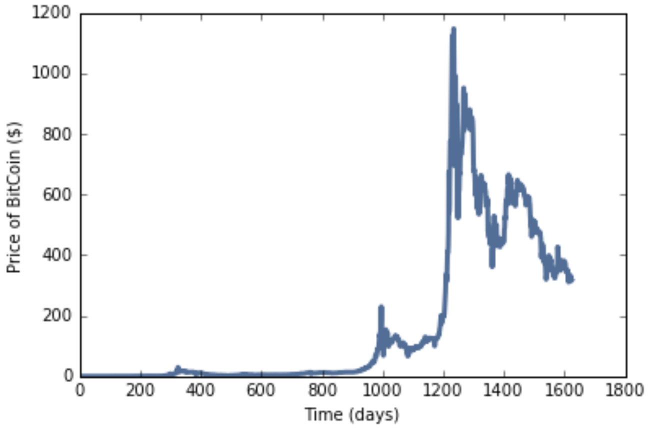
This notebook contains solutions to exercises in Chapter 5: Autocorrelation

In [1]:
```
from __future__ import print_function, division

import thinkdsp
import thinkplot
import thinkstats2

import numpy as np
import pandas as pd

import warnings
warnings.filterwarnings('ignore')

%matplotlib inline
```

Exercise: If you did the exercises in the previous chapter, you downloaded the historical price of BitCoins and estimated the power spectrum of the price changes. Using the same data, compute the autocorrelation of BitCoin prices. Does the autocorrelation function drop off quickly? Is there evidence of periodic behavior?

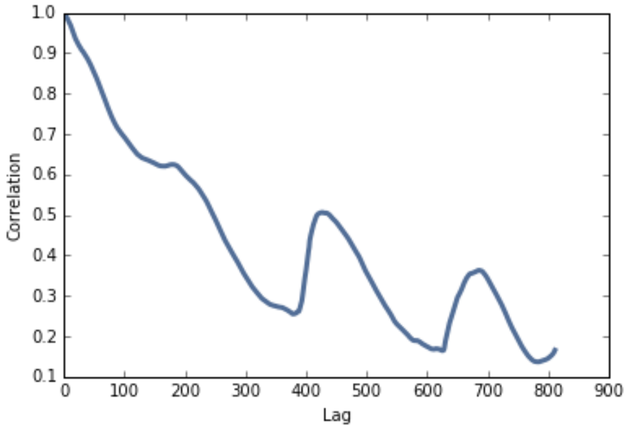
In [2]:
```
df = pd.read_csv('coindesk-bpi-USD-close.csv', nrows=1625, parse_dates=[0])
ys = df.Close.values
```

In [3]:
```
wave = thinkdsp.Wave(ys, framerate=1)
wave.plot()
thinkplot.config(xlabel='Time (days)',
                 ylabel='Price of BitCoin ($)')
```



Here's the autocorrelation function using the statistical definition, which unbiases, normalizes, and standardizes; that is, it shifts the mean to zero, divides through by standard deviation, and divides the sum by N.

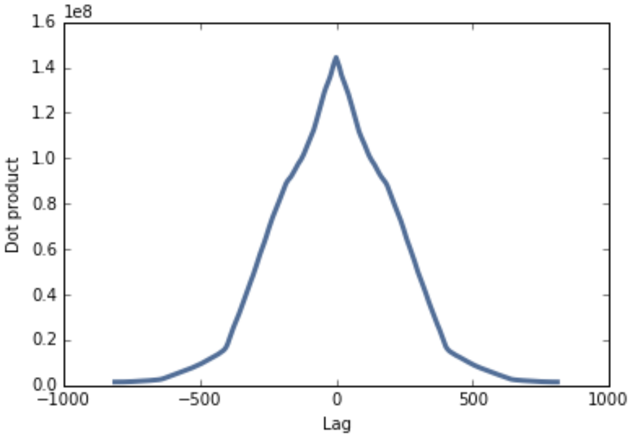
In [4]:
```
from autocorr import autocorr

lags, corrs = autocorr(wave)
thinkplot.plot(lags, corrs)
thinkplot.config(xlabel='Lag',
                 ylabel='Correlation')
```



The ACF drops off slowly as lag increases, suggesting some kind of pink noise. And it looks like there are moderate correlations with lags near 200, 425 and 700 days.

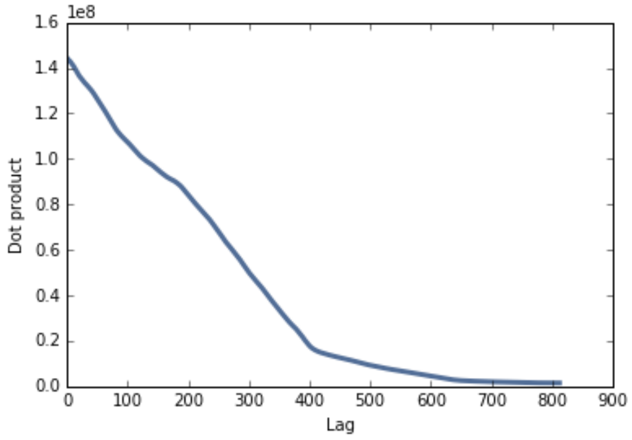
We can compare my implementation of `autocorr` with `np.correlate`, which uses the definition of correlation used in signal processing. It doesn't unbias, normalize, or standardize the wave.

In [5]:
```
N = len(wave)
corrs2 = np.correlate(wave.ys, wave.ys, mode='same')
lags = np.arange(-N//2, N//2)
thinkplot.plot(lags, corrs2)
thinkplot.config(xlabel='Lag',
                 ylabel='Dot product')
```



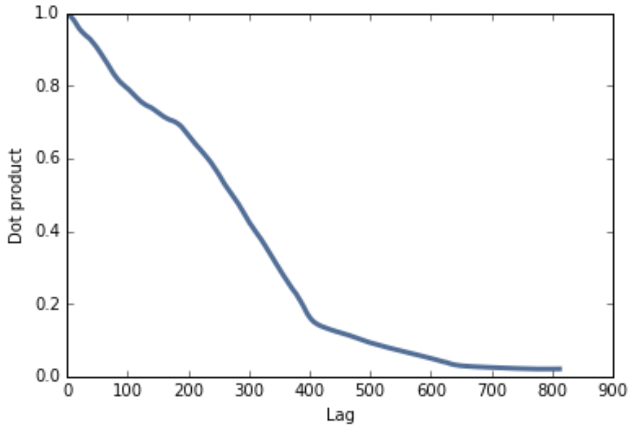
The second half of the result corresponds to positive lags:

In [6]:
```
N = len(corrs2)
half = corrs2[N//2:]
thinkplot.plot(half)
thinkplot.config(xlabel='Lag',
                 ylabel='Dot product')
```



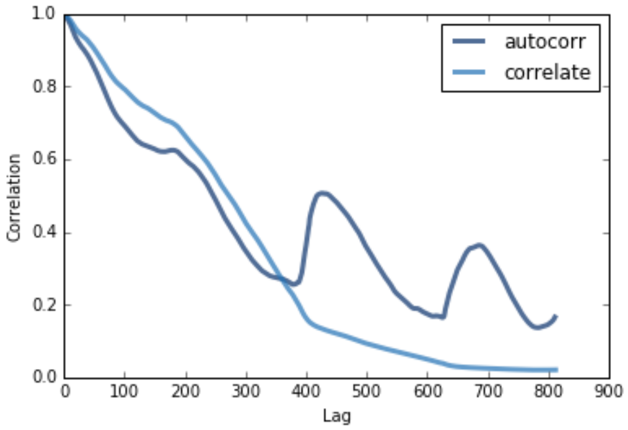
We can standardize the results after the fact by dividing through by `lengths` :

In [7]:
```
lengths = range(N, N//2, -1)
half /= lengths
half /= half[0]
thinkplot.plot(half)
thinkplot.config(xlabel='Lag',
                 ylabel='Dot product')
```



But even after standardizing, the results look very different. In the results from `correlate` , the peak at lag 200 is less apparent, and the other two peaks are obliterated.

In [8]:
```
thinkplot.preplot(2)
thinkplot.plot(corrs, label='autocorr')
thinkplot.plot(half, label='correlate')
thinkplot.config(xlabel='Lag', ylabel='Correlation')
```



I think the reason the results are so different the data look very different in different parts of the range; in particular, the variance changes a lot over time.

For this dataset, the statistical definition of ACF, is probably more appropriate.

Exercise: The example code in `chap05.ipynb` shows how to use autocorrelation to estimate the fundamental frequency of a periodic signal. Encapsulate this code in a function called `estimate_fundamental`, and use it to track the pitch of a recorded sound.

To see how well it works, try superimposing your pitch estimates on a spectrogram of the recording.

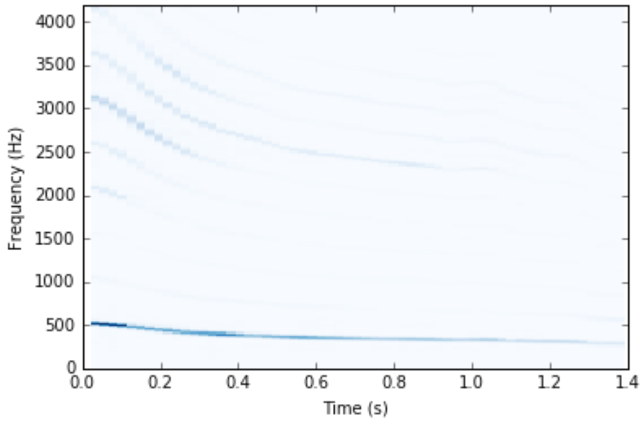
In [9]:
```
wave = thinkdsp.read_wave('28042__bcjordan__voicedownbew.wav')
wave.normalize()
wave.make_audio()
```

Out[9]:

0:00:00 / 13:31:36

I'll use the same example from `chap05.ipynb`. Here's the spectrogram:

In [10]:
```
wave.make_spectrogram(2048).plot(high=4200)
thinkplot.config(xlabel='Time (s)',
                     ylabel='Frequency (Hz)',
                     xlim=[0, 1.4],
                     ylim=[0, 4200])
```



And here's a function that encapsulates the code from Chapter 5. In general, finding the first, highest peak in the autocorrelation function is tricky. I kept it simple by specifying the range of lags to search.

In [11]:
```
def estimate_fundamental(segment, low=70, high=150):
    lags, corrs = autocorr(segment)
    lag = np.array(corrs[low:high]).argmax() + low
    period = lag / segment.framerate
    frequency = 1 / period
    return frequency
```

Here's an example of how it works.

In [12]:
```
duration = 0.01
segment = wave.segment(start=0.2, duration=duration)
freq = estimate_fundamental(segment)
freq
```

Out[12]: 436.63366336633663

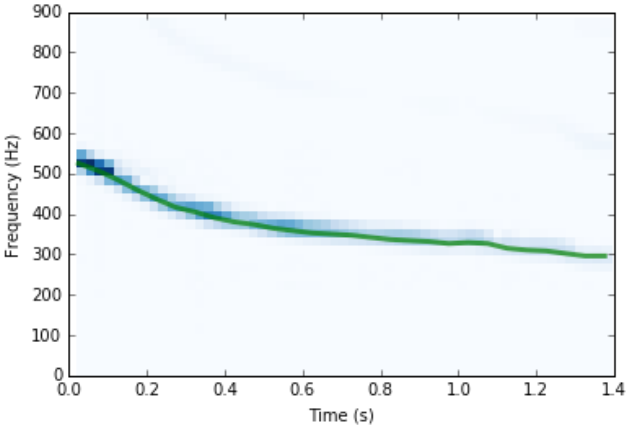
And here's a loop that tracks pitch over the sample.

The `ts` are the mid-points of each segment.

In [13]:
```
step = 0.05
starts = np.arange(0.0, 1.4, step)

ts = []
freqs = []

for start in starts:
    ts.append(start + step/2)
    segment = wave.segment(start=start, duration=duration)
    freq = estimate_fundamental(segment)
    freqs.append(freq)
```

Here's the pitch-tracking curve superimposed on the spectrogram:

In [14]:
```
wave.make_spectrogram(2048).plot(high=900)
thinkplot.plot(ts, freqs, color='green')
thinkplot.config(xlabel='Time (s)',
                 ylabel='Frequency (Hz)',
                 xlim=[0, 1.4],
                 ylim=[0, 900])
```



Looks pretty good!

In [ ]: