```
In [5]: import numpy as np
        import numpy as np
        import pandas as pd
        import scipy as sp
        from pandas import DataFrame, Series
        pd.set_option("display.max_rows", 100)
        pd.set_option("display.max_columns", 400)
        # For Visualization
        import matplotlib.pyplot as plt
        import matplotlib
        matplotlib.rcParams['image.cmap'] = 'viridis'
        %matplotlib inline
        import scipy.stats as stats
        np.random.seed(78)
        matplotlib.style.use('ggplot')
        matplotlib.rcParams['figure.figsize'] = (10, 6)
```

Sometimes physics dictates a functional relationship between two quantities such as velocity and kinetic energy which is given by $E = \frac{1}{2}mv^2$. Sometimes we don't know the relationship, or at least the coefficients, and we need to discover the relationship from measurements.

Suppose that there were two variables x and y with a polynomial relationship

$$y = -0.03x^2 + 0.5x + 1$$

under ideal conditions. We have a bunch of measurements of x and y. Lets say that we have some measurements of x between -30 and +30. To simulate a random set of measurements we are going to use the numpy random library. First we should seed the random library so that we will be able to repeat our experiments even though it is choosing "random" numbers.

```
In [2]: np.random.seed(123)
```

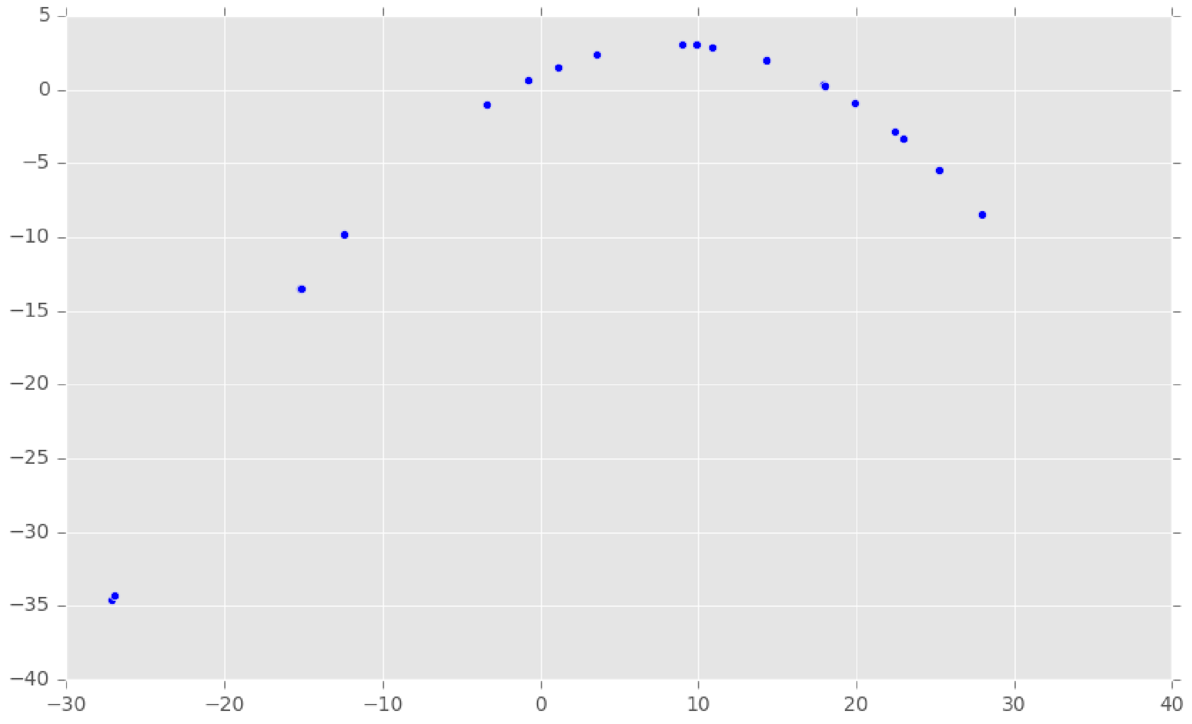
Now lets say we have 20 sample xs, from -30 to +30 randomly chosen. We obtain these samples with the numpy uniform random number generator:

```
In [6]: samples = 20
        xs = sorted(-30 + np.random.random(size=(samples,))*60)
```

Now we apply the functional relationship between x and y to get a set of ys that we will call clean since they are perfect and have no noise.

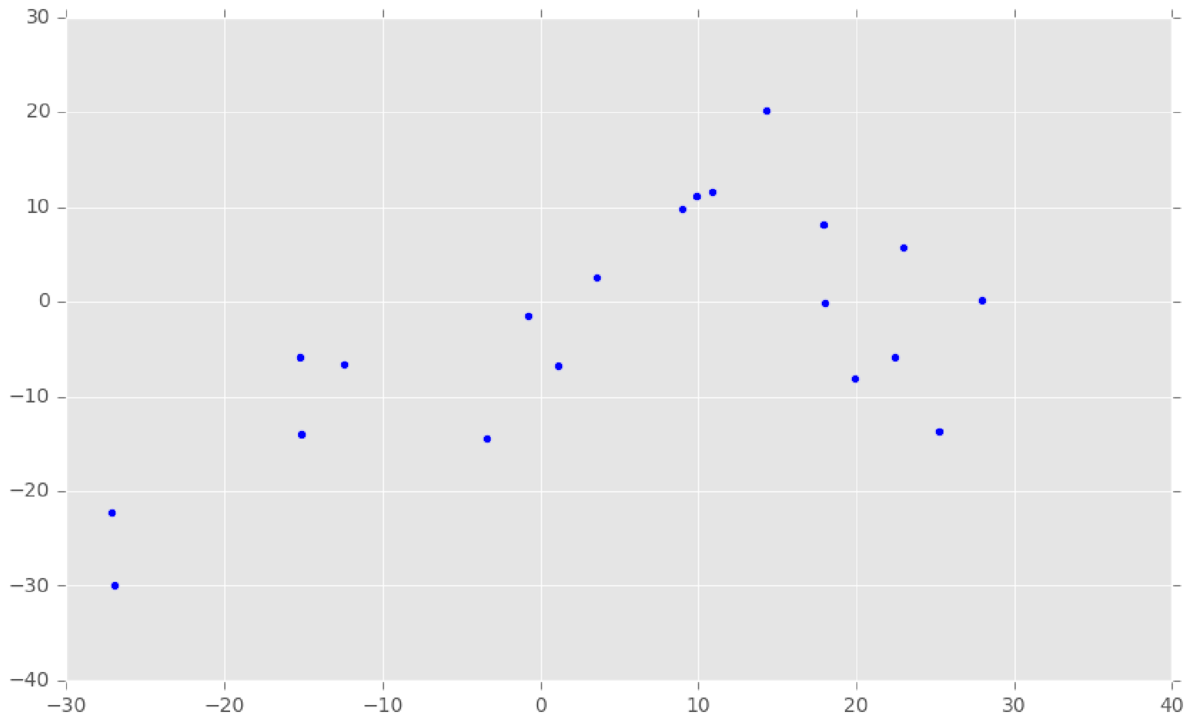
```
In [7]: orig_coeffs= [-0.03,0.5,1]
        ysclean = np.polyval(orig_coeffs,xs)
        plt.scatter(xs,ysclean)
```

```
Out[7]: <matplotlib.collections.PathCollection at 0x119663278>
```



Real data would not look so clean. Usually with real measurements there is "noise". This noise can come from the intrinsic imprecision in the instruments you use to measure. Thermometers may not be perfectly calibrated. Cameras may have dirt on their lenses, or distortion. Noise can also come from other "signals" interfering with what you want to measure. Suppose you are trying to measure the vibrations of a distant earthquake and a truck rolls by. Your seismograph will show vibrations that are a mixture of the earthquake and the truck. To model this messy addition we add normally distributed (distributed via the Gaussian "bell curve") values. Here we "noise-up" y

```
In [9]: ys = ysclean + 8*np.random.randn(len(ysclean))
        plt.scatter(xs,ys)

Out[9]: <matplotlib.collections.PathCollection at 0x11a231320>
```



At this point xs and ys are simulated measured data. We will now pretend somebody gave us these xs and ys as measurements from an instrument. We will forget that we knew the coefficients of the polynomial equation. We will even forget that we knew it was a quadratic. Lets just assume we know, or hypothesize that the relationship between x and y is some kind of polynomial. From the graph our first guess might be that there is a linear relationship between x and y. In order to find the line we apply linear regression. Since there are more than two points, and they don't exactly line on a line, there is no single line, no choice of m and b for the equation $y_{\text{line}} = m * x + b$ which will intersect for all the measured points.

The points don't lie on a line because for one, we secretly know that the right model is a quadratic (degree 2). In addition and more importantly, there is noise so that even if the right model were a line the data would not exactly lie on it. Even so, we can find the best line, the line for which the error $\varepsilon = y_{\text{line}} - y_{\text{measured}}$ is as small as possible. Then the root mean square error (RMSE) is the square root of the mean of all the squares of the errors

$$RMSE = \sqrt{\frac{1}{n}\sum_i = 1^N \varepsilon_i^2}$$

. Finding the parameters of the line that make this error as small as possible is called least squares fit.

Here is how we do it in numpy. We use the polyfit routine with degree set to 1 for a line fit.

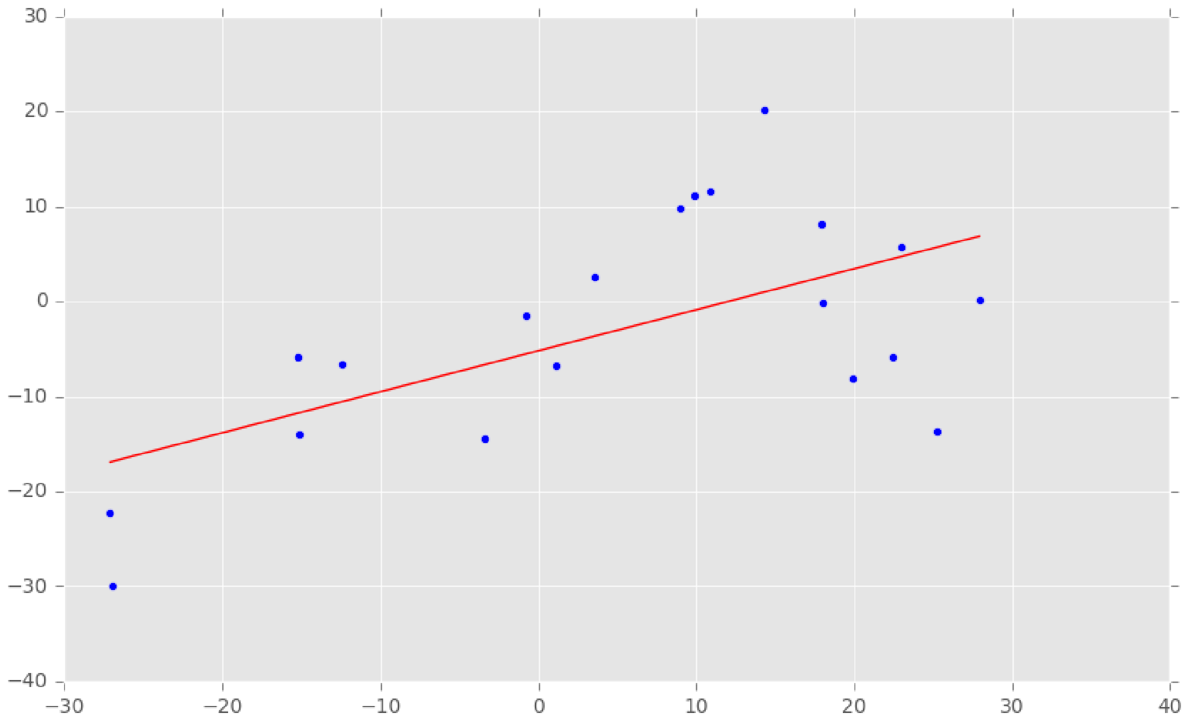
```
In [20]: linear_degree = 1
         linear_coeffs = np.polyfit(xs,ys,linear_degree)
```

Now that we have coefficients for a line, lets draw that line by draw in red along some points u and v. Polyval is just a numpy function that computes a polynomial with the given coefficients.

```
In [21]: u = np.linspace(min(xs),max(xs),100)
         v = np.polyval(linear_coeffs,u)
```

We will plot the red line superimposed over blue scatter of the xs and ys points we are treating as measured data.

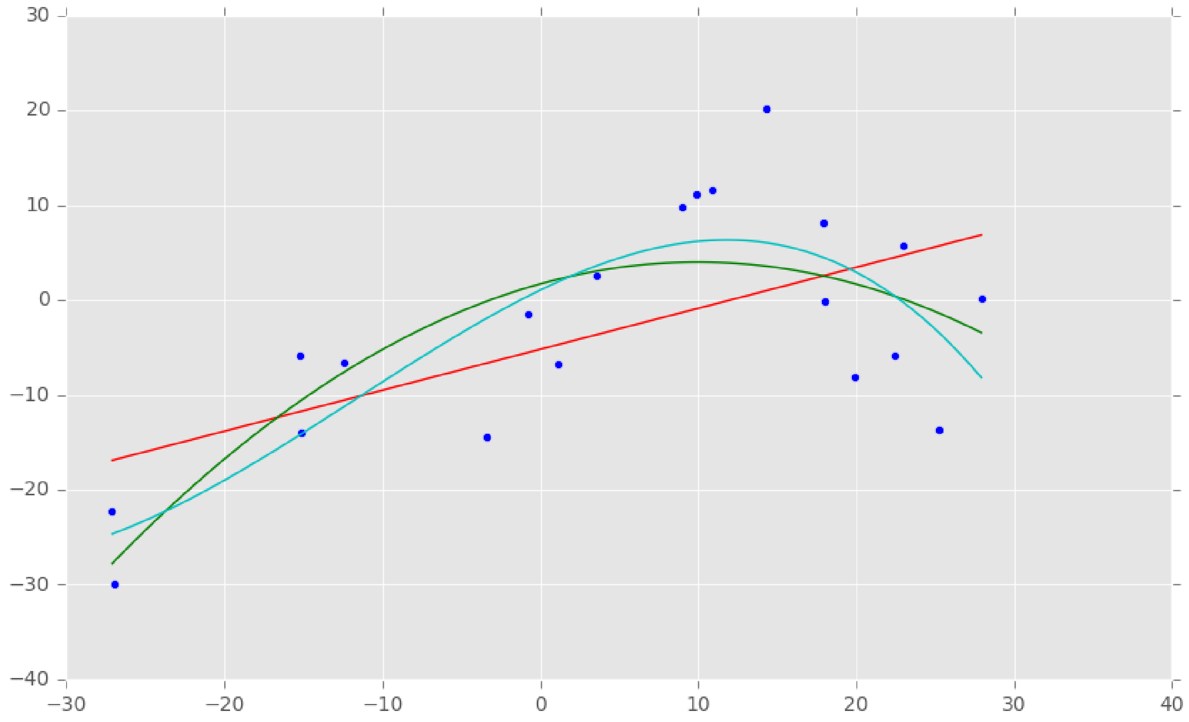
```
In [22]: plt.scatter(xs,ys)
         plt.plot(u,v,'-r')

Out[22]: [<matplotlib.lines.Line2D at 0x11aabcb00>]
```



```
In [23]: linear_degree = 2
         linear_coeffs = np.polyfit(xs,ys,linear_degree)
         v2 = np.polyval(linear_coeffs,u)
         linear_degree = 3
         linear_coeffs = np.polyfit(xs,ys,linear_degree)
         v3 = np.polyval(linear_coeffs,u)
```

```
In [24]: plt.scatter(xs,ys)
         plt.plot(u,v,'-r')
         plt.plot(u,v2,'-g')
         plt.plot(u,v3,'-c')
```

```
Out[24]: [<matplotlib.lines.Line2D at 0x11ad35978>]
```



It is always possible that if we use a higher order polynomial, we could get a better fit. Lets try to fit the points using higher polynomials and see what the error looks like. First we will get the coefficients for a polynomial fit for degrees 0 through 7.

```
In [57]: max_degree = 12
         list_of_coeffs =[np.polyfit(xs,ys,degree) for degree in np.arange(0,max_
         degree)]
```

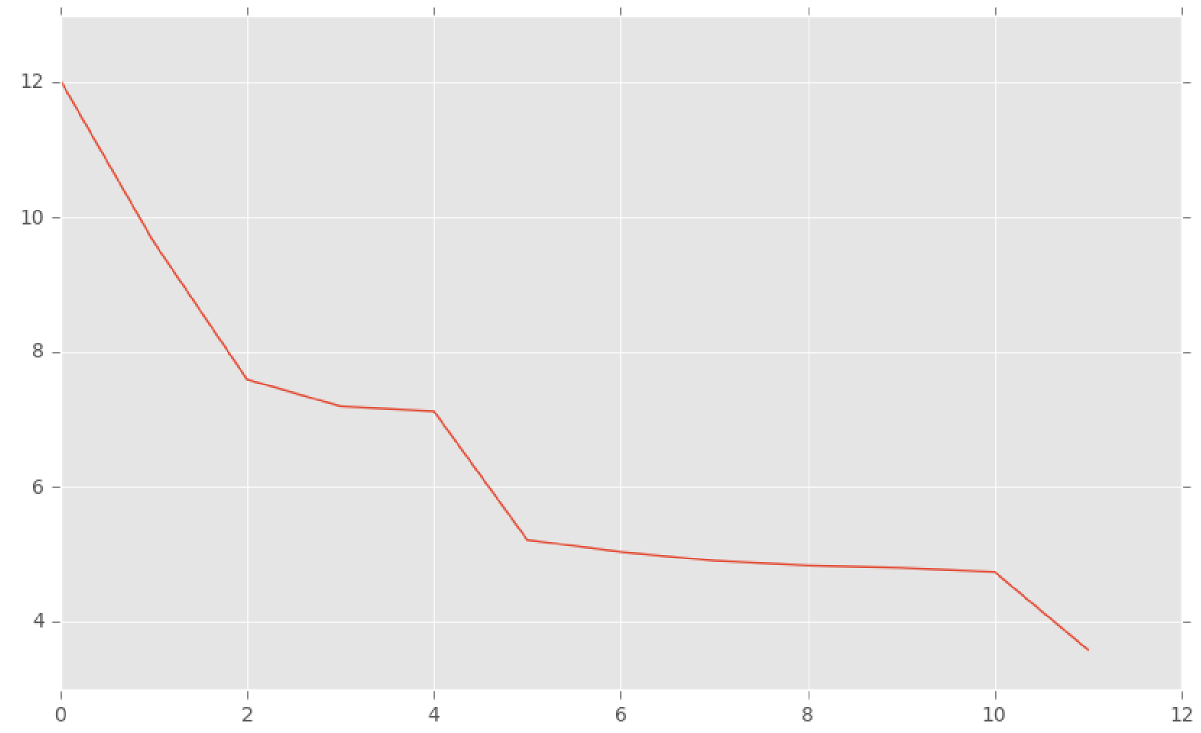
We can compute the error of fit, the RMSE, by seeing by applying polyval to xs to get the value of y predicted by the fit polynomails, and comparing that to the actual ys. The differences are squared, the means taken, and then the square root taken. Lets write a function to compute the RMSE.

```
In [58]: def RMSE(predicted_y,actual_y):
             return np.sqrt(((predicted_y-actual_y)**2).mean())
```

Now we will use the function to see how as degree varies the error of fit varies.

In [59]:
```
error_of_fit = [ RMSE(np.polyval(coeffs,xs),ys) for coeffs in list_of_co
effs]
plt.plot(range(0,max_degree),error_of_fit)
```

Out[59]:
```
[<matplotlib.lines.Line2D at 0x11c9560b8>]
```



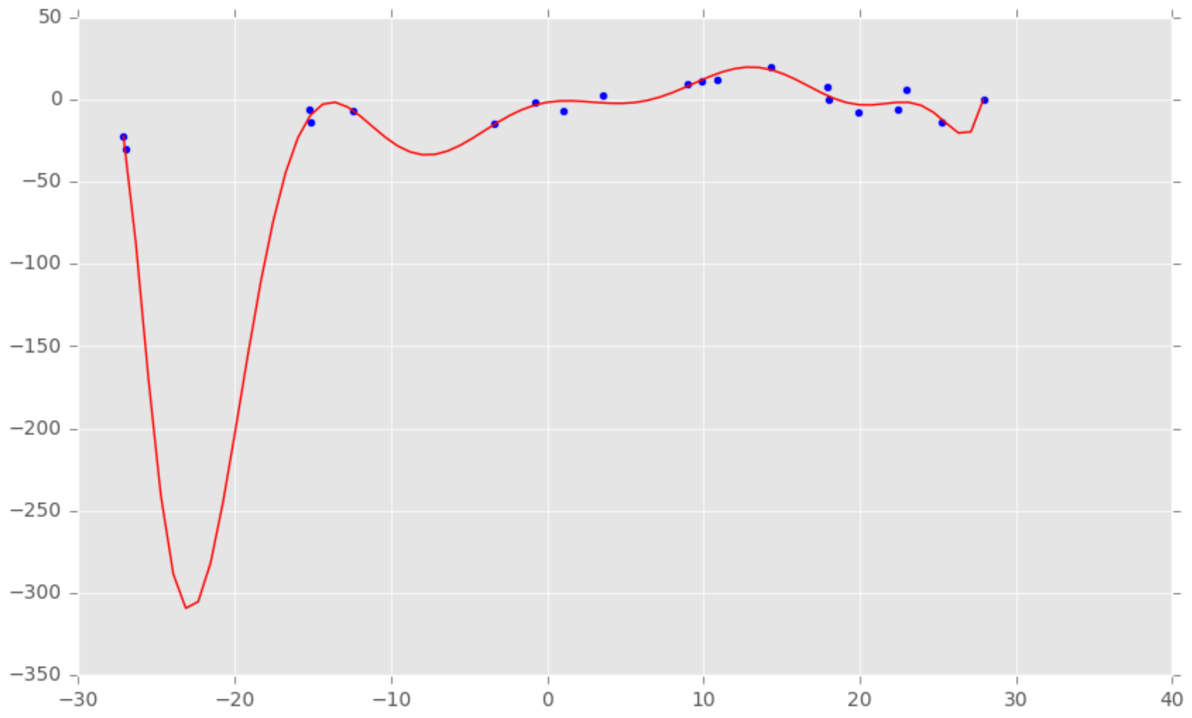
This looks like great news! The higher the degree, the better the fit. The problem is that as the degree goes up the fit, while modeling the points well, produces a polynomial that likely has nothing to do with the original process we were trying to model. Lets just look at degree 7 for example.

In [61]:
```
poly_coeff = list_of_coeffs[-1]
```

In [62]:
```
u = np.linspace(min(xs),max(xs),70)
v = np.polyval(poly_coeff,u)
```

```
In [63]: plt.scatter(xs,ys)
         plt.plot(u,v,'-r')
```

```
Out[63]: [<matplotlib.lines.Line2D at 0x11c77a6a0>]
```



As we get closer to fitting the points, the polynomial will create wiggles outside the data points. This is because least squares "punishes" the polynomials for not being close to the data but not for having spurious wiggles in between the dots. The higher the degree we allow, the closer the red curve will be to the dots but the more the fit curve will have twists and turns that aren't meaningful. This is called "overfitting." This means we can't just see how close the curve fits the points to measure how well our model is doing. Instead what we will do is fit a polynomial with some of our points but test the fit on points we didn't use to fit. Imagine if we had some sample points in the above 7th degree polynomial graph with $x = 25$. The value y would probably be around 0. If we tested the degree 7 polynomial it would do very badly since it drops down to -50 there. Hence if we use a measure based on testing on points we didn't fit on, when the degree gets too high, and we start overfitting, the error should go up. One way to fit and test on different points is using a technique called "leave one out". The idea here is that for a given polynomial degree, we fit on all the points except one: we leave one out. We start out by leaving out the first point, then the second, then the third. Each time we compute the polynomial, plug the x value of the point we left out back in and compare it with the known y value. The RMSE of the differences is our error measure. First lets create a leave one out function that takes an index and returns a list with that element omitted. It is a little messy because we need to be doing this with plain old python lists, not numpy arrays.

```
In [48]: def leave_1_out(data,leave_out_index):
             # This makes sure it is a plane old python list
             data = list(data[:])
             return data[:leave_out_index] + data[leave_out_index+1:]
```

Now we will fit over and over leaving a different point out each time with the same polynomial degree. Lets make this a function too. It takes a degree, the xs, and ys data as input and outputs a list of coefficients.

```
In [53]: def leave_one_out_coeffs_list(degree,xs,ys):
             result = [polyfit(leave_1_out(xs,leave_out_index),
                       leave_1_out(ys,leave_out_index),
                       degree)
                       for leave_out_index in xrange(0,len(xs))]
             return result
```

Lets write a function that computes the error value applied to the missing point, for a given degree by looping over all possible left out points.

```
In [56]: def evaluate(degree,xs,ys):
             # We don't want to fit if the degree is too high. The polynomial
             # blows up if the degree is higher than the number of points to fit.
             # lets stop before we get there
             if len(xs) < degree +1:
                 return np.inf
             list_of_cfs = leave_one_out_coeffs_list(degree,xs,ys)
             predicted_ys = [polyval(cf,x) for cf,x in zip(list_of_cfs,xs)]
             return RMSE(predicted_ys ,ys)
```

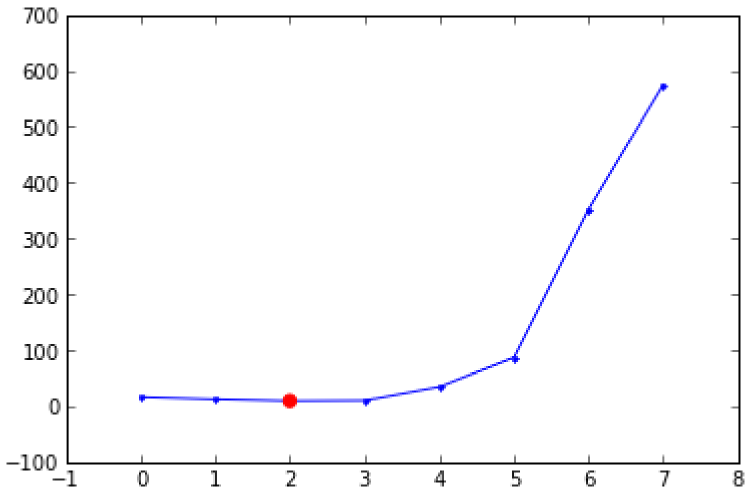
At this point lets evaluate the error for each degree.

```
In [57]: leave_1_out_eval = [evaluate(degree,xs,ys) for degree in xrange(8)]
```

Here we use the argmin function of a numpy array to find the index of an array, where that row is as small as possible. We are doing this because it will tell us the degree that minimizes the error.

```
In [62]: degree_min = np.array(leave_1_out_eval).argmin()
```

```
In [63]: min_val = leave_1_out_eval[degree_min]
In [64]: # Plot the errors for each degree (blue dots)
         plot(xrange(8), leave_1_out_eval,marker='.',c='b' );
         # Plot the minimizeing degree as a big red dot.
         scatter([float(degree_min)],[min_val],color='r',s=40.0,zorder=100);
```



Now that we know we get the best results for degree 2 we can just fit using degree = 2 to find the coefficients.

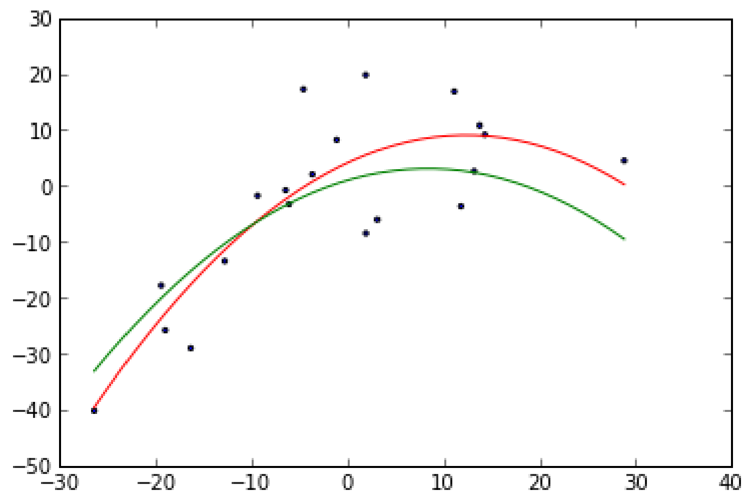
```
In [65]: final_coeffs = polyfit(xs,ys,2)
```

```
In [66]: print final_coeffs

         [-0.03239761  0.7993425   4.14046664]
```

```
In [68]: print orig_coeffs

         [-0.03, 0.5, 1]
```

```
In [69]: u = np.linspace(min(xs),max(xs),40)
         vfin = polyval(final_coeffs,u)
         vorig = polyval(orig_coeffs,u)
```

```
In [70]: scatter(xs,ys,marker='.',c='b')
         plot(u,vfin,'r-')
         plot(u,vorig,'g-')

Out[70]: [<matplotlib.lines.Line2D at 0x108071d90>]
```



```
In [ ]:
```