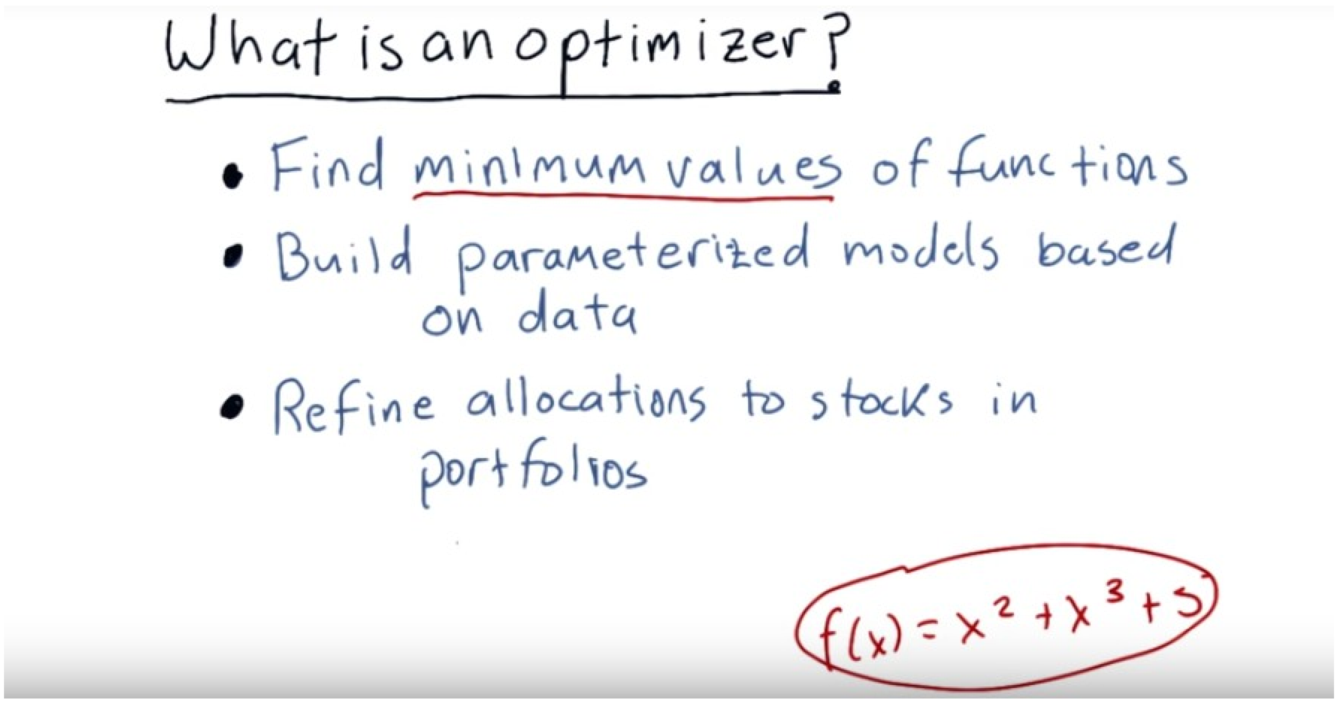
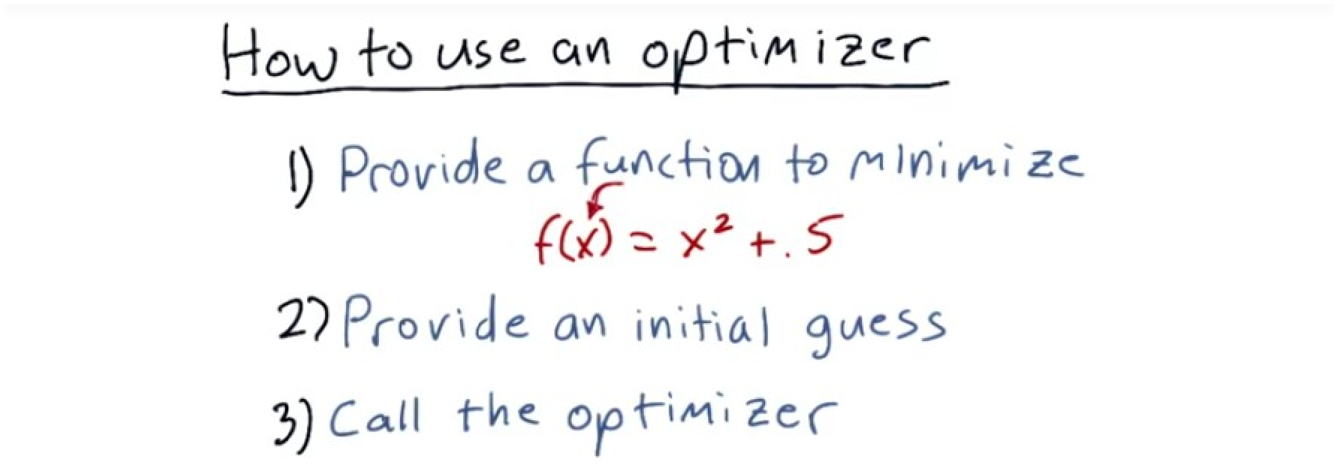

What is an optimizer?

- Find minimum values of functions
- Build parameterized models based on data
- Refine allocations to stocks in portfolios

$f(x) = x^2 + x^3 + 5$

1. In this lesson we're going to look at optimizers. Optimizers sound scary, but they're really cool and really fun. I'm going to show you all kinds of neat things you can do with optimizers. What is an optimizer? An optimizer is an algorithm that can do the following things. Optimizers can be used to find minimum values for functions. So say you have a function like f(x)=x2+x3+5 or something like that, an optimizer can find. For what value of x is this overall function minimized? Another thing that optimizers can do is find the parameters for parameterized models from data. So we might have some data from some experiment, and we can use optimizers to find a polynomial fit to that data. And that is actually one thing we are going to learn in this lesson. Finally, we can use an optimizer to refine allocations to stocks in portfolios. What does that mean? Well, that means for instance, you can decide what percentage of funds should be allocated to each stock using an optimizer.


How to use an optimizer

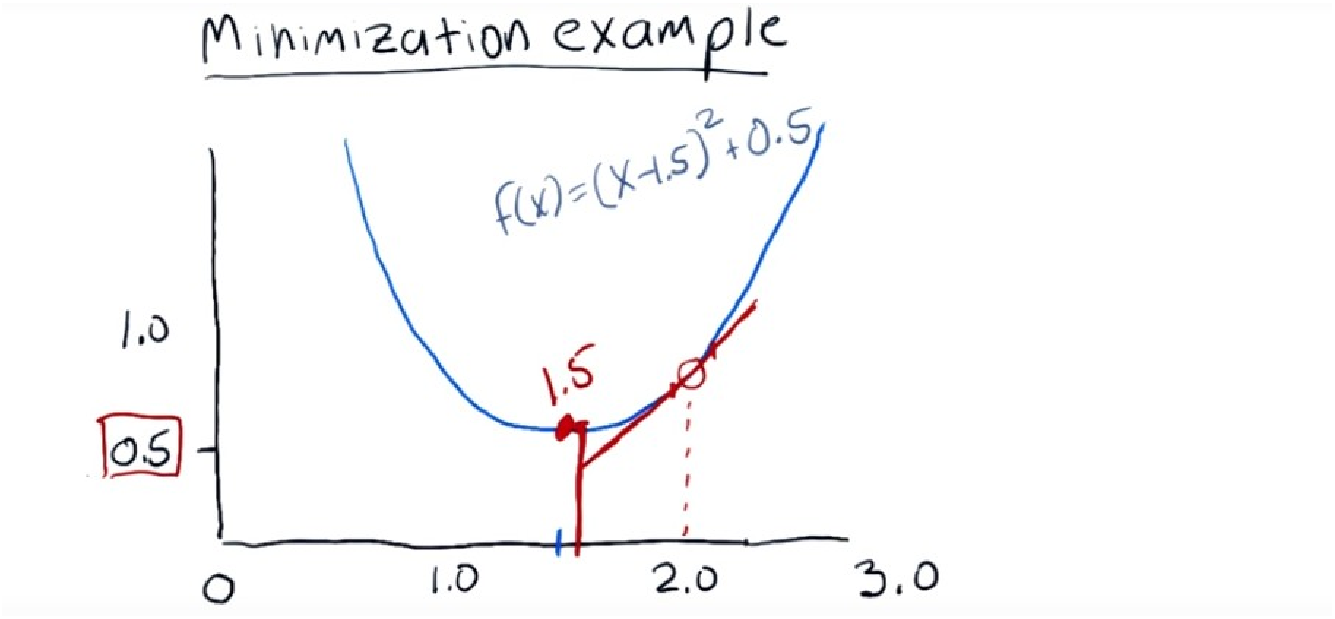
1) Provide a function to minimize

$f(x) = x^2 + .5$

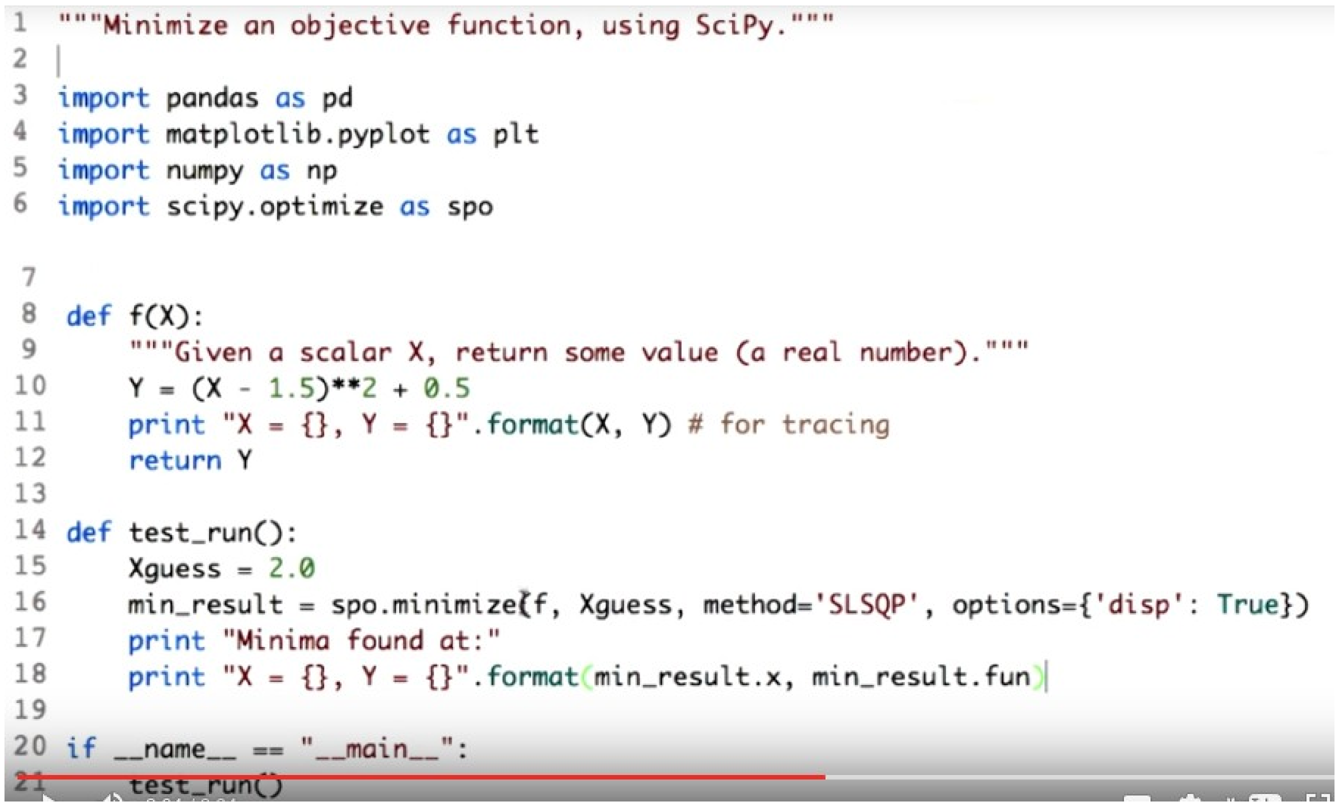
2) Provide an initial guess

3) Call the optimizer

How do we use an optimizer? It's really just as simple as three key steps. First thing you need to do is define a function that you want to minimize. As an example you might use something like f of x is equal to x square plus point five. You define that in Python and then the minimizer will call this function many, many times as it tries to find the minimum values for x that causes this function overall to be smallest. You also need to start with an initial guess for x that you think might be close to the solution to the problem. If you don't really know, then you can choose a random value or just some standard value. But then the optimizer starts with that guess and it repeatedly calls a function, tests different values, and narrows in on the solution. Finally, you call the optimizer with these parameters and stand back while it searches for the minimum.

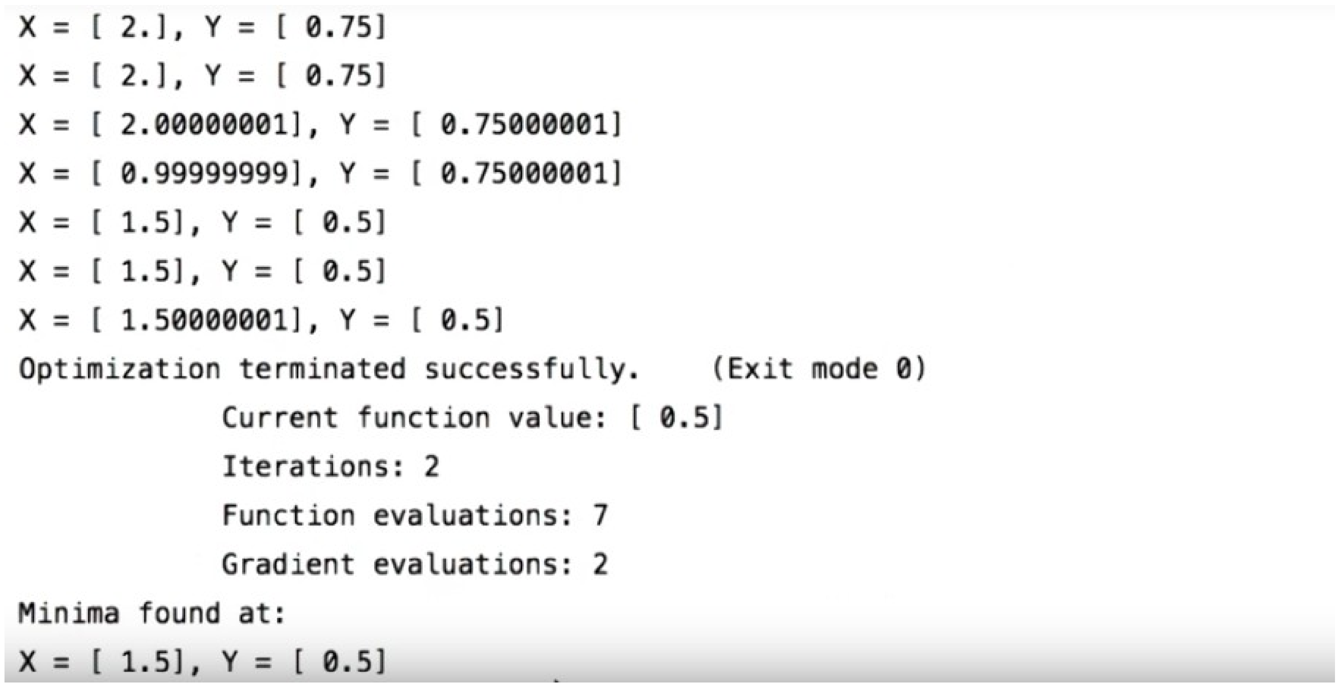


2. Let's take a look at this function, f(x) is equal to x minus 1.5 squared plus 0.5. That function is a parabola that looks something like this. It's centered horizontally at 1.5, and its minima is here at 0.5. Now, the minimizer doesn't know that. We can tell it by looking at the equation, but the minimizer has to figure it out on its own. So let's suppose we tell it, hey minimizer, why don't you start with a guess of 2.0 and see if you can figure out from there what it is? So the minimizer says, okay, I'll give it a go. [LAUGH] And here's what it does. First thing it does is it checks the value at 2.0, it turns out that that's about 0.75. It then tests the value nearby, say here and here. And it finds out that this equation has a slope about like that, at this point. Now, it's trying to minimize, and so what it does is it marches downhill, it's called gradient descent. And it tries another value down along that slope. Gets a particular value here, tries another one, and so on. And eventually, it narrows and it discovers that 1.5 is the value for x at the minima. And the value of y there is 0.5. Now, the example I gave you for sort of marching down this gradient descent is one method. There are many variations on that method, that different kinds of minimizers use. And Scifi, the library that we're using, has many of those options. And you can choose different ones according to your taste. We're going to stick with one particular approach through our examples here. But you ought to experiment and try some of the other ones as well.



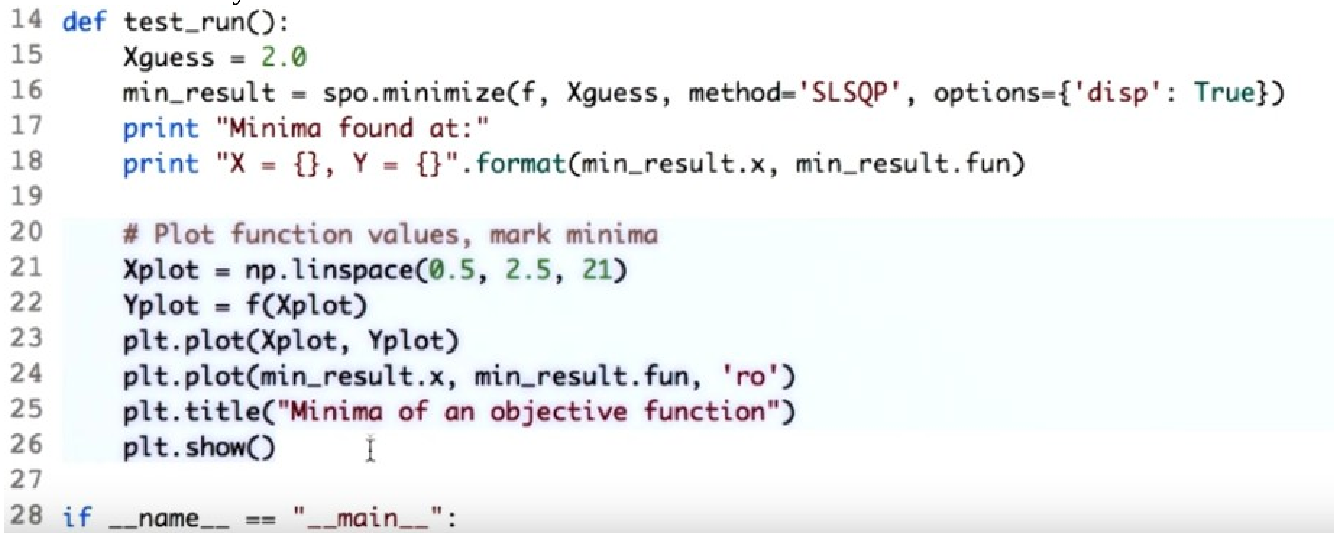
```python
"""Minimize an objective function, using SciPy."""

import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
import scipy.optimize as spo

def f(X):
    """Given a scalar X, return some value (a real number)."""
    Y = (X - 1.5)**2 + 0.5
    print "X = {}, Y = {}".format(X, Y) # for tracing
    return Y

def test_run():
    Xguess = 2.0
    min_result = spo.minimize(f, Xguess, method='SLSQP', options={'disp': True})
    print "Minima found at:"
    print "X = {}, Y = {}".format(min_result.x, min_result.fun)

if __name__ == "__main__":
    test_run()
```

3. Let's try that same example function now in Python code. So up here we have our normal imports, here is where we define the function and again we're simply using X-1.5 squared +0.5. Now within this function we're going to go ahead and print what the value is when we get called. It just is a little bit handier so that we can see what exactly is going on. But you don't have to have that of course. And then we return y. Now this is going to be the function that we're going to Ask SciPy(讀作 saipai), or in particular the optimizer, to minimize for us. And by the way, we've included this optimize package as spo. So scipy.optimize as spo. This is our call now to the optimizer or the minimizer. Before we call it we first set our guess value to be 2.0. And we're using the function minimize so we call spo.minimize. F, that's our function here, so we're saying minimizer, please minimize find the minimum for this function. X guess is our guess. Method is, we're directing minimize to use a particular minimizing algorithm. We'll talk a little bit about that a little bit later. But this is one of those particular algorithms that happens to work pretty nicely. We send it one more option here, disp, which is True. Which means we just want it to be verbose(冗長的) about things that it discovers. Anyways, that's it. That calls the minimizer. The minimizer repeatedly calls our function and finds the minimum value, then it prints out those results.



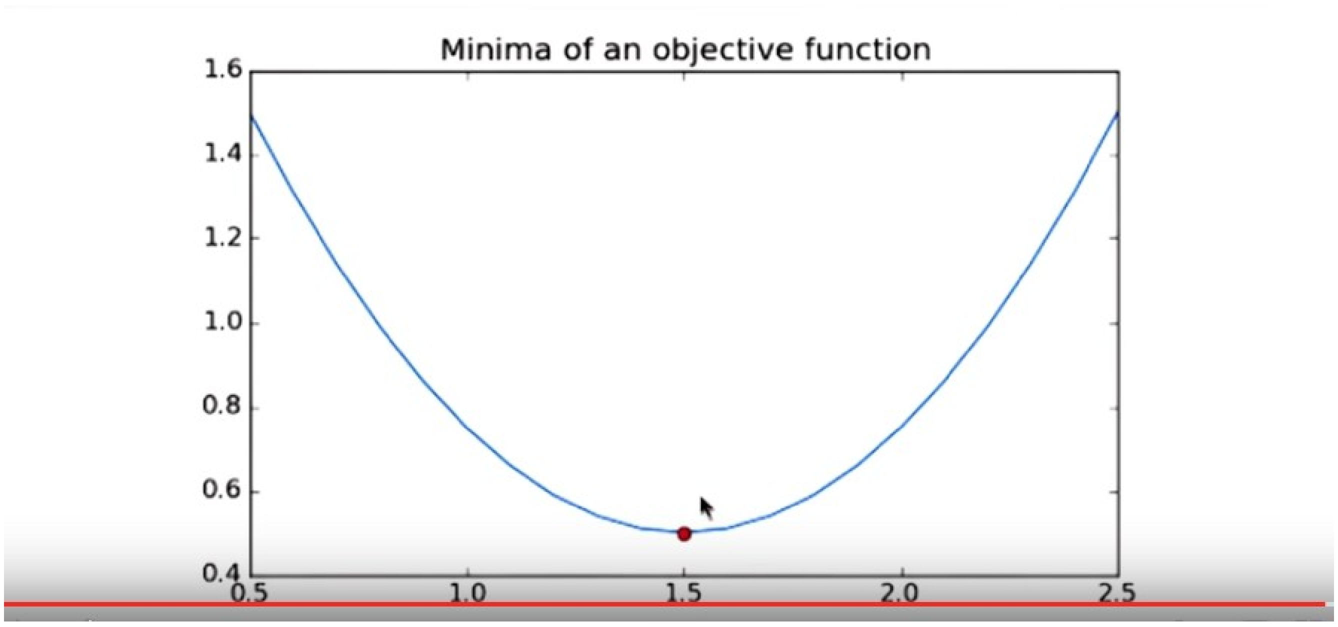
```
X = [ 2.], Y = [ 0.75]
X = [ 2.], Y = [ 0.75]
X = [ 2.00000001], Y = [ 0.75000001]
X = [ 0.99999999], Y = [ 0.75000001]
X = [ 1.5], Y = [ 0.5]
X = [ 1.5], Y = [ 0.5]
X = [ 1.50000001], Y = [ 0.5]
Optimization terminated successfully.    (Exit mode 0)
            Current function value: [ 0.5]
            Iterations: 2
            Function evaluations: 7
            Gradient evaluations: 2
Minima found at:
X = [ 1.5], Y = [ 0.5]
```

Let's try a test run and see what happens. Remember, in our function that we wanted to minimize, we explicitly printed X and Y. So here you can see each time it gets called it prints these values out. And so the minimizer is repeatedly calling that function f and it's printing these things out. So it gets called initially with an X of 2 and it discovers that the value is 0.75. Then a value slightly greater than 2, a value slightly less than 2. And the minimizer very quickly converges on 1.5 as the answer, and here it prints out those values and finds the minima at 1.5 with a value of 0.5 there. So pretty efficient and effective discovery of the minimization.



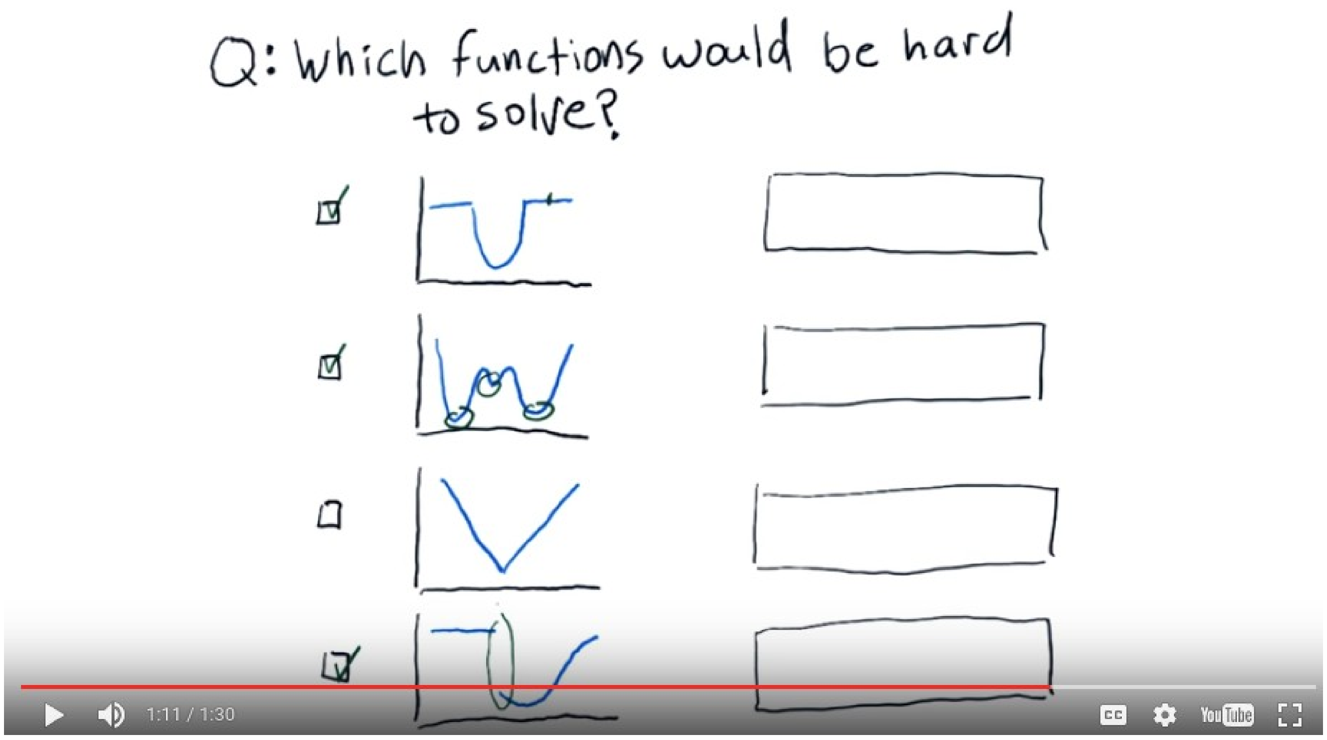
```python
def test_run():
    Xguess = 2.0
    min_result = spo.minimize(f, Xguess, method='SLSQP', options={'disp': True})
    print "Minima found at:"
    print "X = {}, Y = {}".format(min_result.x, min_result.fun)

    # Plot function values, mark minima
    Xplot = np.linspace(0.5, 2.5, 21)
    Yplot = f(Xplot)
    plt.plot(Xplot, Yplot)
    plt.plot(min_result.x, min_result.fun, 'ro')
    plt.title("Minima of an objective function")
    plt.show()

if __name__ == "__main__":
```

I added a few more lines of code here which I'll highlight, merely to plot the answer, so all the rest of the code is the same.

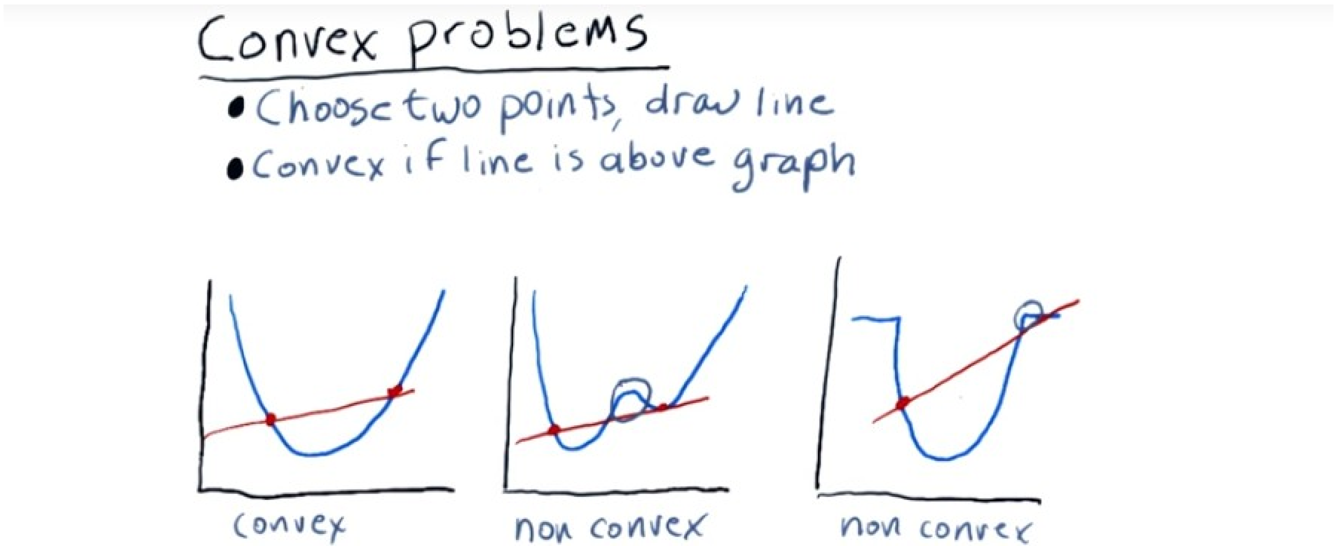


But let's take a look now if we plot it as well. So, same result as before but nice plot with our minima identified right here. So, that is how to code up a minimizer, it's really very easy and very powerful. Let's look at it a little bit further.

4. Now that you know how minimizers or optimizers work, think a little bit about what might be hard for them to solve. So I'm going to show you four example function shapes. And I want you to consider whether these would be hard or easy for the minimizer to solve. Here are four function shapes to look at. If you think that one would be hard for the minimizer we just talked about, check the box next to it, and tell us why. Type out a reason in the text box. All right, have at it.

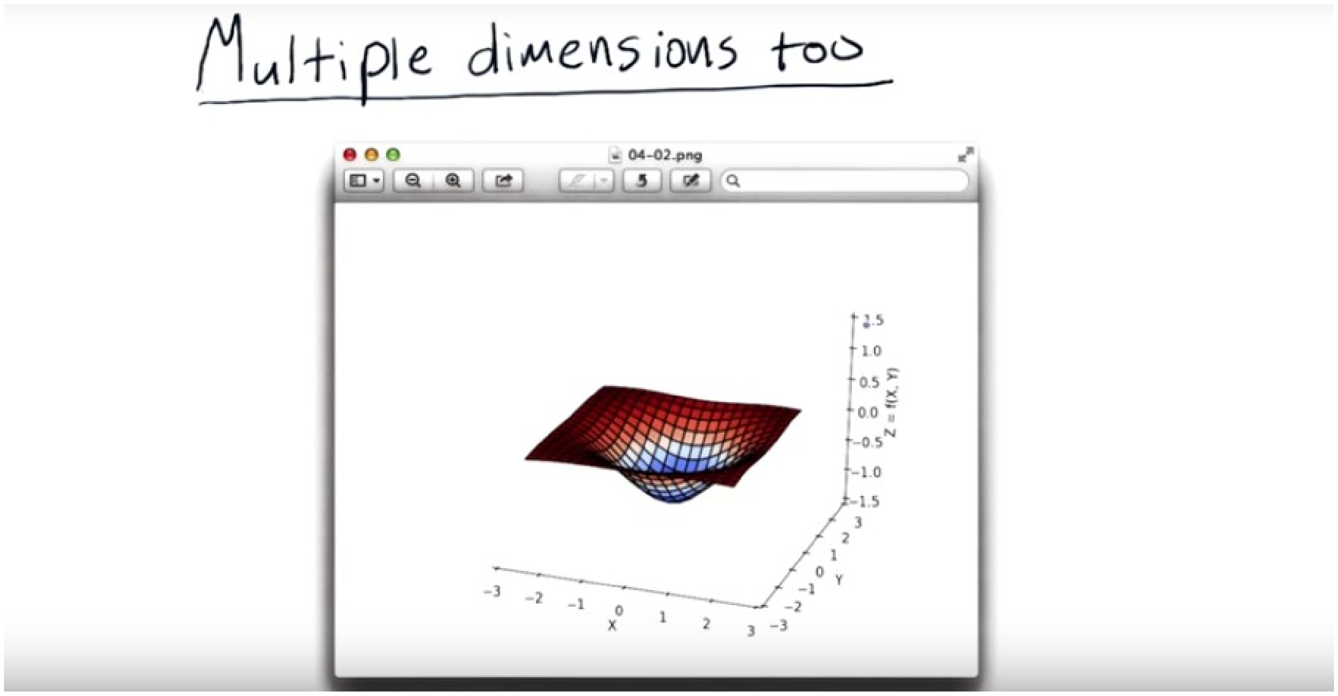


5. Most of these are hard. This one's hard. This one's hard, and this one, and let me explain why. This one (第一個) is hard because of this flat area here and here. Suppose the minimizer tested this point here and then tried on either side. It wouldn't be able to find any gradient to follow, so, it wouldn't know which direction to go. This one (第二個) is difficult for at least two reasons. One is it has several local minima that aren't necessarily the global minima. So, it might iterate and find, say, this as a minima. But notice that actually there's this other two that are actually smaller. And then if these two have exactly the same value, turns out we have two global minima. So, those sorts of conditions are tough for these minimizers to solve. This one (第四個) is challenging A, because of this flat area, but also because of this discontinuity. So, four examples, three of them would be hard for our minimizer to solve. Now, I'm not saying that these are not solvable by optimizers. In fact, there are optimizers that can solve these problems with varying degrees of success. And they're likely to find a minima, just not guaranteed to find the minima.

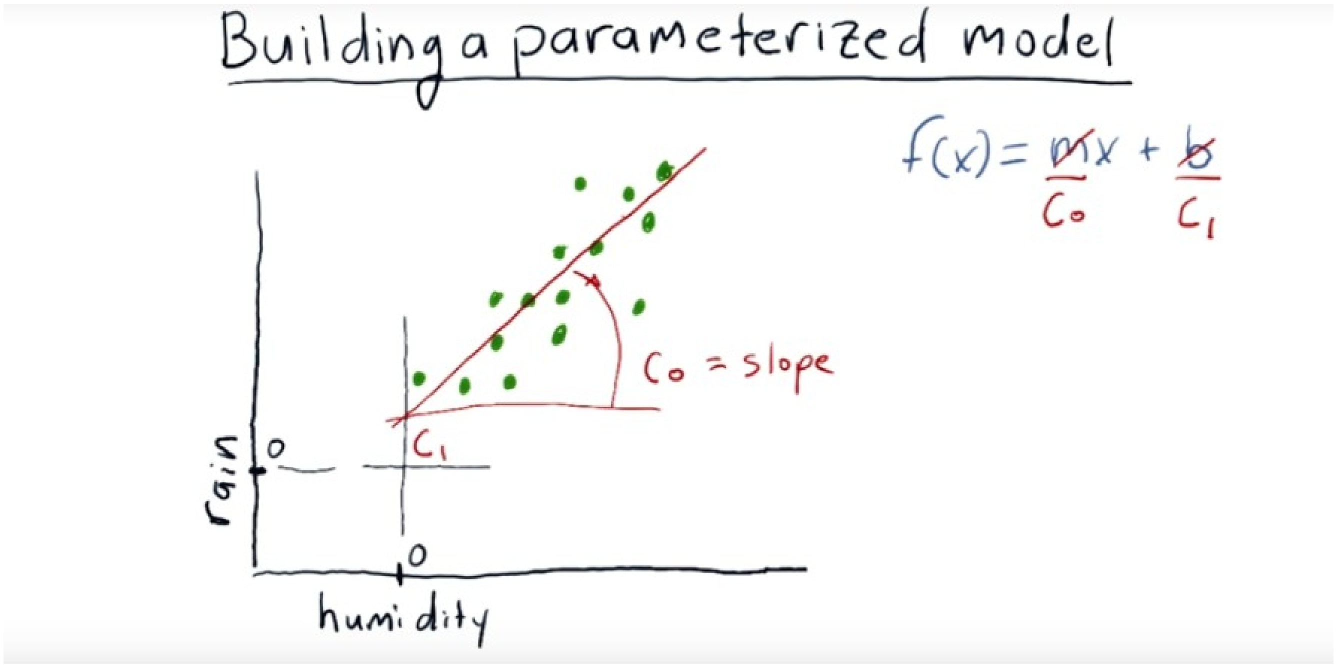

Convex problems
• Choose two points, draw line
• Convex if line is above graph
convex
non convex
non convex

6. While we're on the topic of problems that are easy or hard for optimizers to solve, let's talk for a moment about a particular class of problems that are indeed the most easy for these types of algorithms to solve. And those are called convex problems. Here's the formal definition of a convex function. I'm going to read it to you from Wikipedia. And then I'll show you what it means on these graphs here. A real valued function f of x defined on an interval is called convex if the line segment between any two points on the graph of the function lies above the graph. A lot of words there. Let me show you what that means more easily. First step, choose two points and draw a line between them. Now, for each of these lines, if the line is above the graph, everywhere between those two points, then the function is convex between those points. So for this function, yes, it's convex because the line is above the graph everywhere. In fact, any two points you chose on this graph, we'll have that property. So this function is convex everywhere, at least where we're looking at. Here, notice that this part of the graph lies above the line. So this is non convex. Similarly, this one, we've got this region here that lies above the line, so this one is also non convex. And this one is of course convex. So a couple things to observe here, some properties that emerged. One is in order for the function to be convex, it has to have only one local minima. And in other words, that local minima is the global minima. This one fails for that reason. We also can't have any flat regions that essentially don't have any slope downward. Now, if the function you're trying to find a minima for is convex, then these algorithms will find the minima quickly and easily. But again, there are algorithms that can still find the minima for more complicated examples like these. But they require a little bit of randomness and they aren't necessarily guaranteed

to find the global minima.

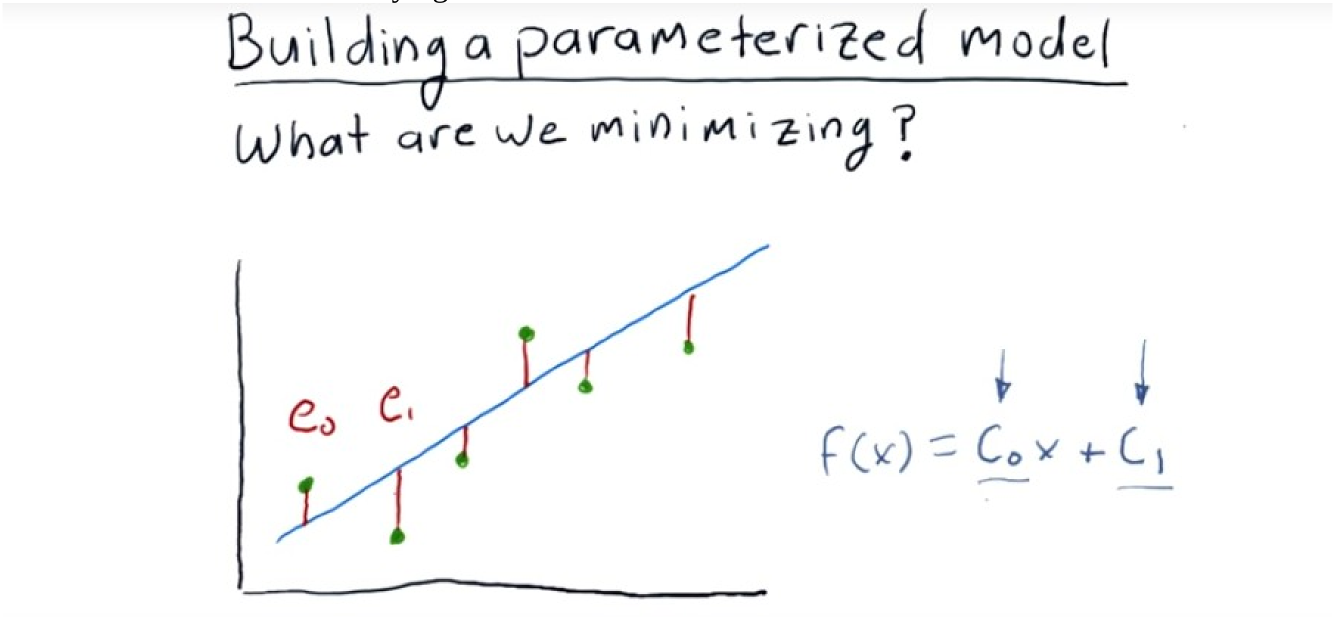


So far, we've been looking at functions that just have one dimension in x. So for instance, the parabola that we looked at. It's just as easy for these optimizers to work in multiple dimensions. Here's an example of a function that has two dimensions in x. It still has its y result. But the minimizers can solve these problems with gradient descent just as easily. So instead of just one dimension, we can have one, two, three, four, as many as we'd like.



7. Now we're going to do something really cool. I'm going to show you how to build a parameterized model from data. What do I mean by parameterized model? This is an example of a parameterized

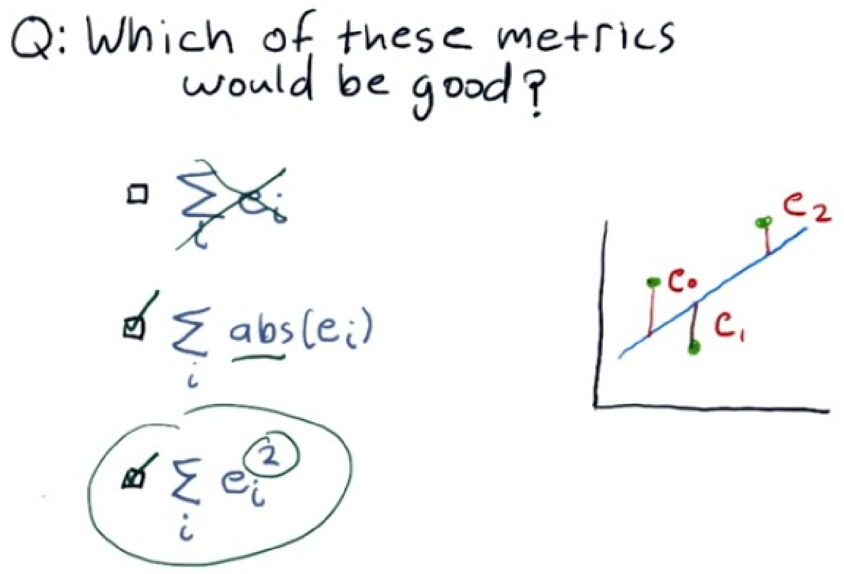
model that you're probably familiar with from algebra. It's a function of x and it has these two parameters, m and b. In fact, as you're probably aware, this is the equation of a line. So m and b are the parameters of that line. Now for convenience in our code instead of using m and b, I'm going to use C0 and C1, just to be consistent. Let me motivate this with an example. Let's suppose we have some data from an experiment. Now this can work for many sorts of experiments, but for now, let's assume we've taken some measurements of humidity, and we've observed on those particular days we measured the humidity how much it rains. So each dot here represents one day and one sample of data. So on this date, it was this humid, and it rained that much. Now we probably have lots more data, one for each day. When we look at this data, we see there's a kind of relationship here. And our intuition is maybe that it could be fitted by a line. Just sort of by eyeballing it, looks like the line might look about like that, and so our parameters here coefficient 0 is equivalent to the slope here, and coefficient 1 is the y intercept. So our task is to find C sub 0 and C sub 1 that provide the equation for this line that best fits the data. The question here is, how do we reframe this problem so that it makes sense for our minimizer? What is it we're trying to minimize?



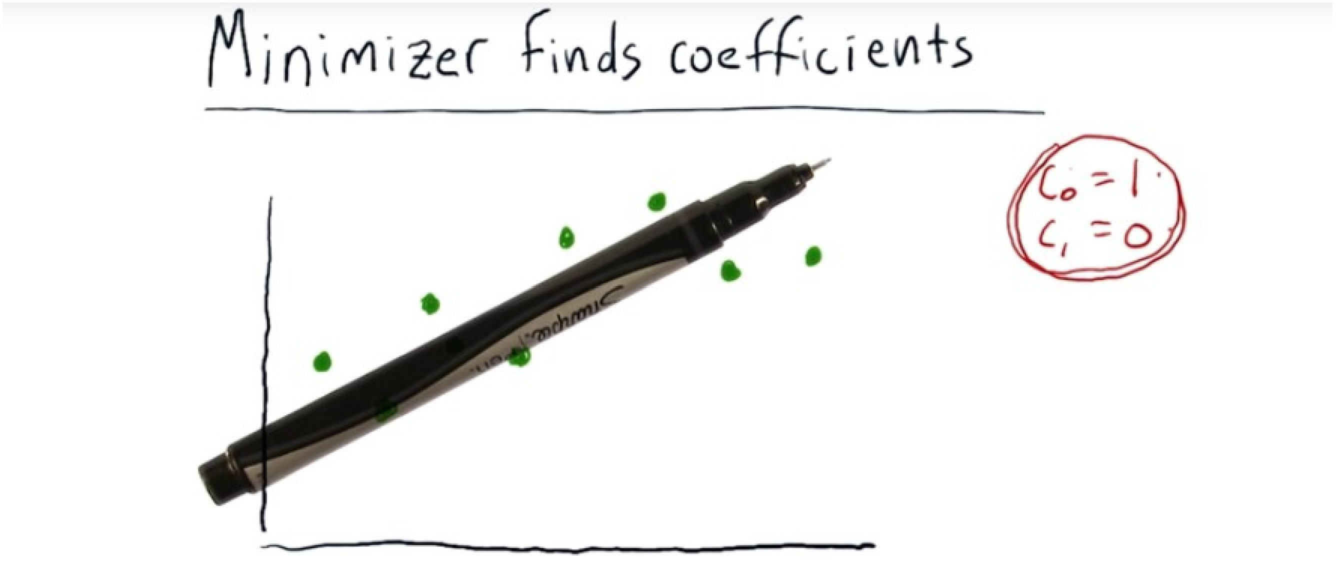
Building a parameterized model
What are we minimizing?
e_0
e_1
$f(x) = \underline{C_0} x + \underline{C_1}$

So restating the problem, suppose we have our original data points here, and we're trying to discover the equation of a line that best fits those points. Suppose this blue line is a candidate line and we want to evaluate it. Is this good or bad? So the equation for that line is, our first coefficient times x plus the second coefficient. And what the minimizer is going to do is it's going to vary this C0 and C1 to try and minimize something. And so we have to come up with an equation that gets lower in value as this line better fits the data. What should we use for that equation? So here's one step towards solving this problem. We can take a look at each one of our original data points and observe how far away it is from this line that we're evaluating. Let's call each of these distances e. So e sub 0 is that one, e sub 1 is that one. Can we come up with an equation in terms of e or error that gets us to this solution?

8. Here's a quiz to get you thinking about that. So again, e is the error at each point. In other words, this is our original data point. This is the line that we're hypothesizing might be a good solution. And we can test how far off our model or our line is at each one of these points and measure that as e. So which of these formulae might be a good overall error measure? There could be more than one.

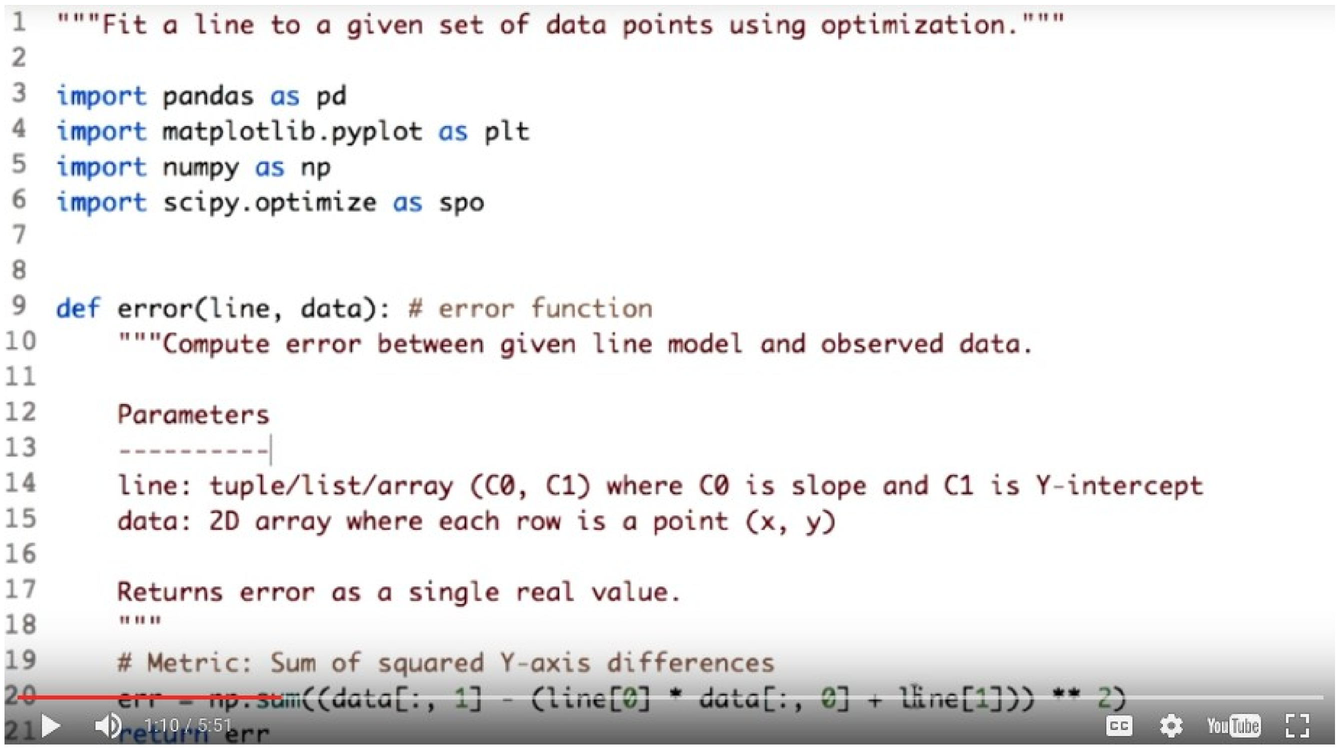


9. So these two are reasonable answers. The reason that this one is not a good answer is because some of these e's may be negative. In other words, this one is negative, these two are positive, but you could end up with a negative error if you just added them up. You can fix that by adding absolute value or by squaring it. This measure here is one of the more famous one. Of course it's squared error.

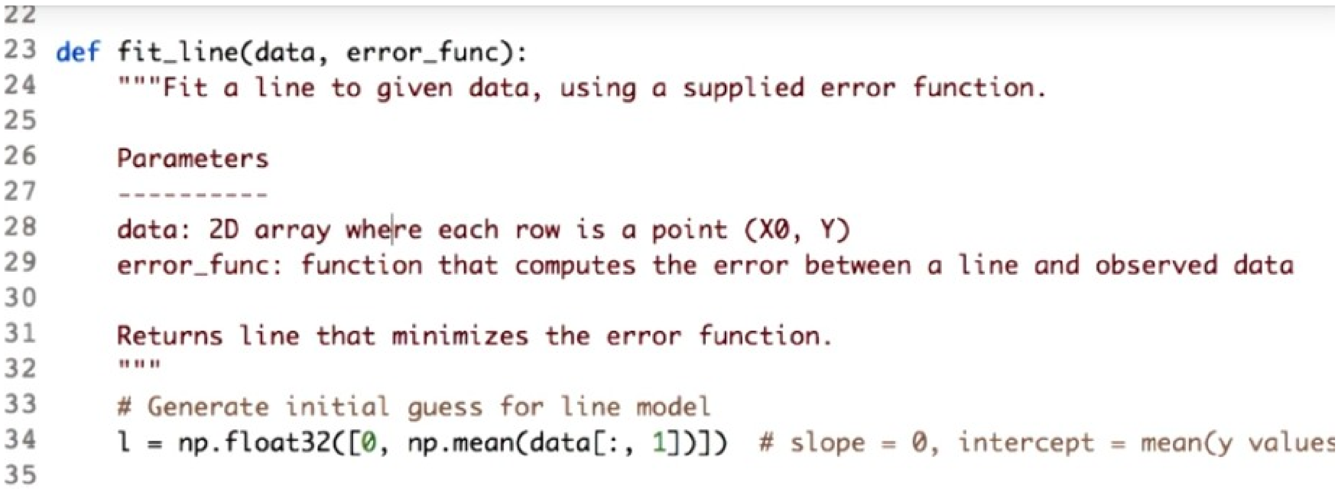


10. Let's step through this now with an example of how a minimizer would try to find the coefficients of a line that best fits this data. So keep in mind that we have to give the minimizer an equation that it has to minimize. And what we're going to give it is that error metric. In fact we used squared error. So we might guess an initial C0 and C1 and that would be a line like this, and we would give that to the minimizer and let it go. So it would measure the error with this particular line, it would fiddle with these values a little bit and see how much the error changed, try a new set of values see how that works. And eventually it's going to iterate, and eventually it's going to settle on what it thinks is the best solution. So key points here are that we express the problem for the minimizer as a minimization problem and we give it the equation to minimize as the error. And then, what it finds now instead of x is it finds the values for these coefficients. So, let me show you how to do that now in code.

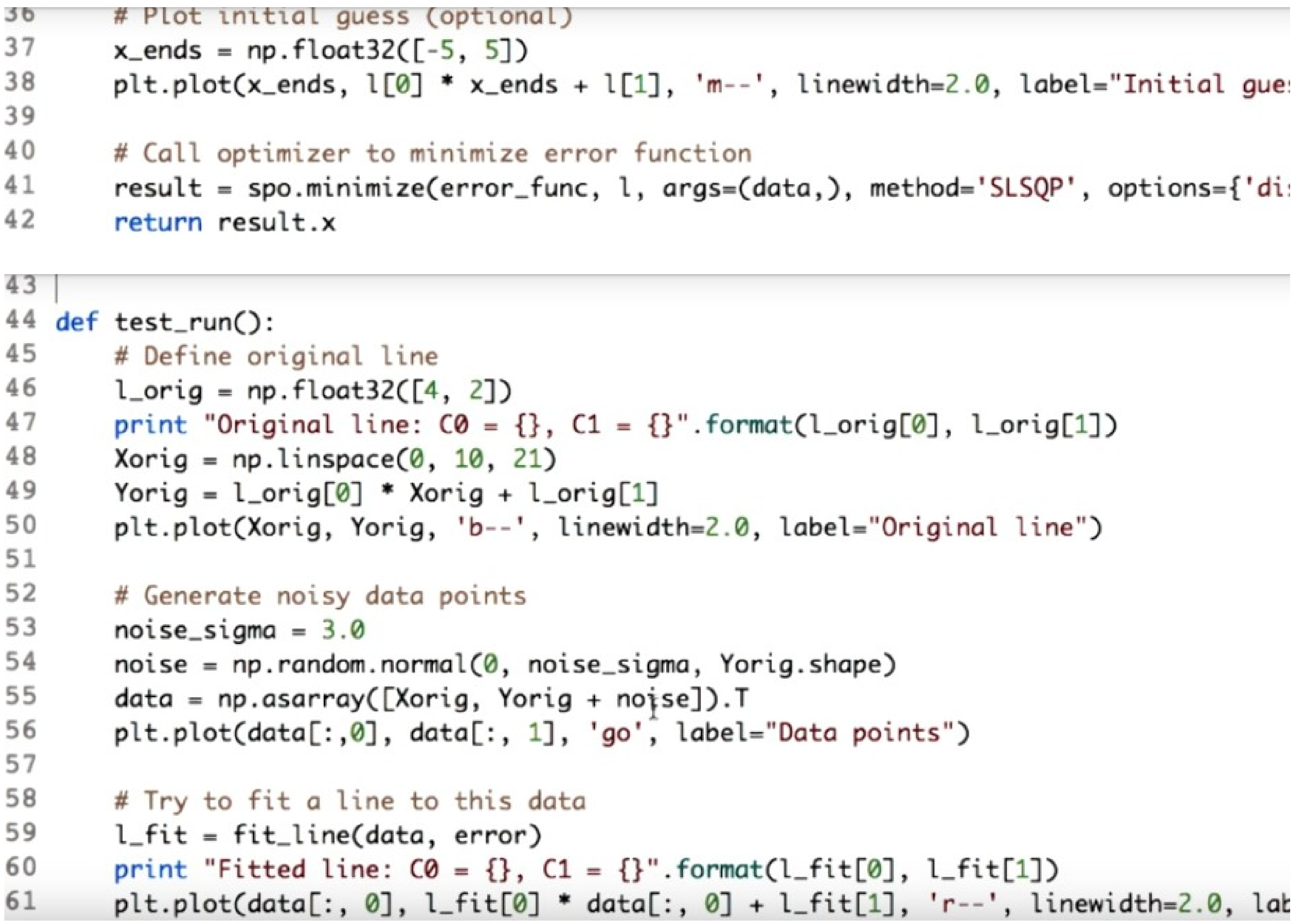
以下(至本文件結束)的內容都沒甚麼意思, 湊時間而已:



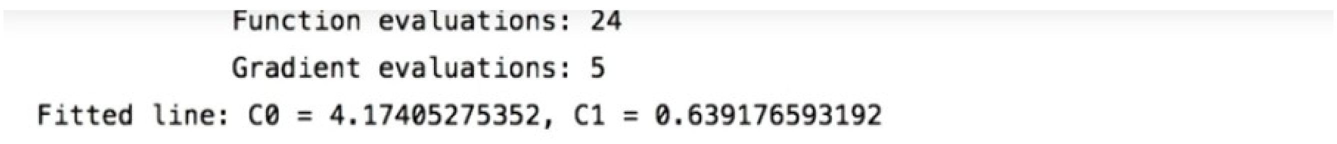
```
"""Fit a line to a given set of data points using optimization."""

import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
import scipy.optimize as spo


def error(line, data): # error function
    """Compute error between given line model and observed data.

    Parameters
    ----------
    line: tuple/list/array (C0, C1) where C0 is slope and C1 is Y-intercept
    data: 2D array where each row is a point (x, y)

    Returns error as a single real value.
    """
    # Metric: Sum of squared Y-axis differences
    err = np.sum((data[:, 1] - (line[0] * data[:, 0] + line[1])) ** 2)
    return err
```



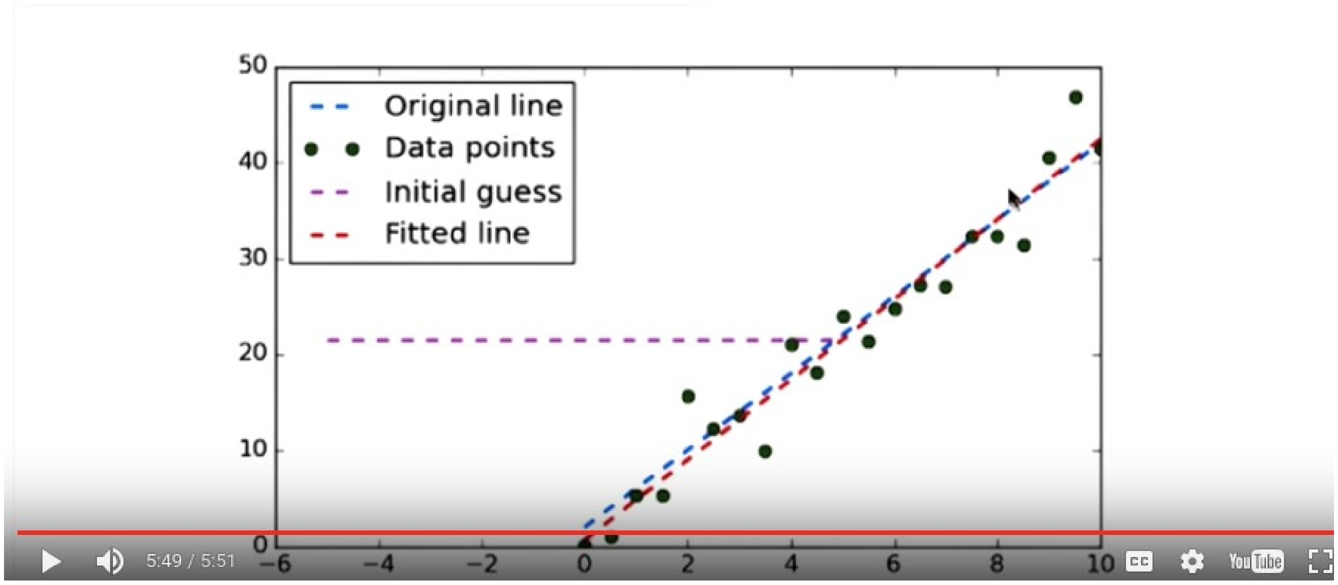
```

def fit_line(data, error_func):
    """Fit a line to given data, using a supplied error function.

    Parameters
    ----------
    data: 2D array where each row is a point (X0, Y)
    error_func: function that computes the error between a line and observed data

    Returns line that minimizes the error function.
    """
    # Generate initial guess for line model
    l = np.float32([0, np.mean(data[:, 1])])  # slope = 0, intercept = mean(y values

```



~~~
    # Plot initial guess (optional)
    x_ends = np.float32([-5, 5])
    plt.plot(x_ends, l[0] * x_ends + l[1], 'm--', linewidth=2.0, label="Initial gue

    # Call optimizer to minimize error function
    result = spo.minimize(error_func, l, args=(data,), method='SLSQP', options={'di
    return result.x

def test_run():
    # Define original line
    l_orig = np.float32([4, 2])
    print "Original line: C0 = {}, C1 = {}".format(l_orig[0], l_orig[1])
    Xorig = np.linspace(0, 10, 21)
    Yorig = l_orig[0] * Xorig + l_orig[1]
    plt.plot(Xorig, Yorig, 'b--', linewidth=2.0, label="Original line")

    # Generate noisy data points
    noise_sigma = 3.0
    noise = np.random.normal(0, noise_sigma, Yorig.shape)
    data = np.asarray([Xorig, Yorig + noise]).T
    plt.plot(data[:,0], data[:, 1], 'go', label="Data points")

    # Try to fit a line to this data
    l_fit = fit_line(data, error)
    print "Fitted line: C0 = {}, C1 = {}".format(l_fit[0], l_fit[1])
    plt.plot(data[:, 0], l_fit[0] * data[:, 0] + l_fit[1], 'r--', linewidth=2.0, lab
```]
~~~

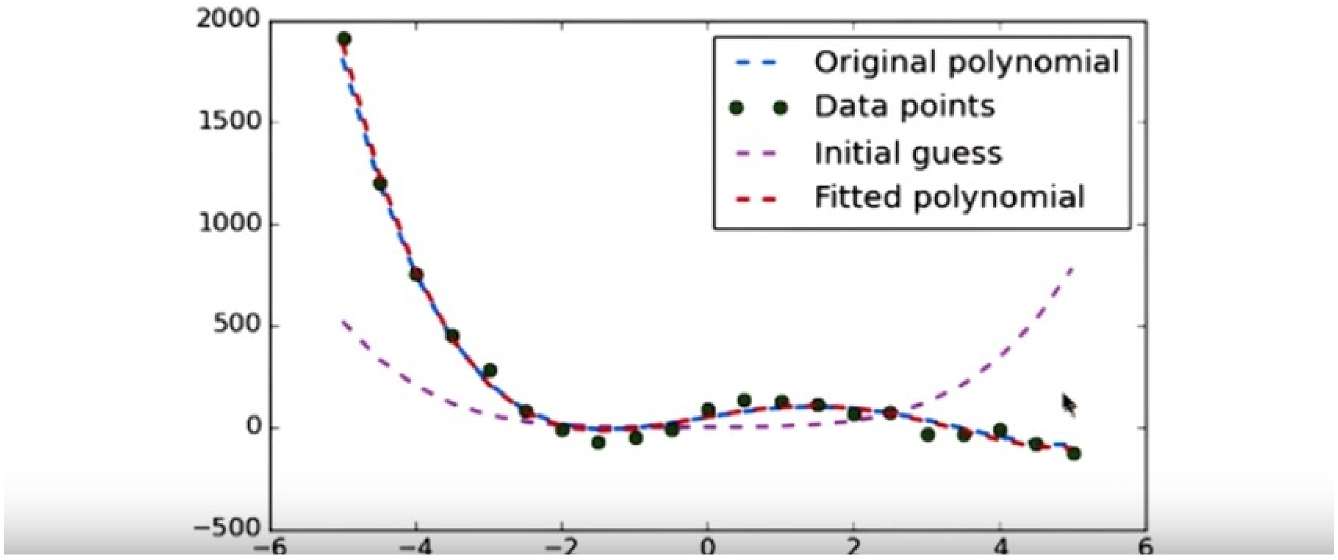


```
                  Function evaluations: 24
                  Gradient evaluations: 5
 Fitted line: C0 = 4.17405275352, C1 = 0.639176593192
```

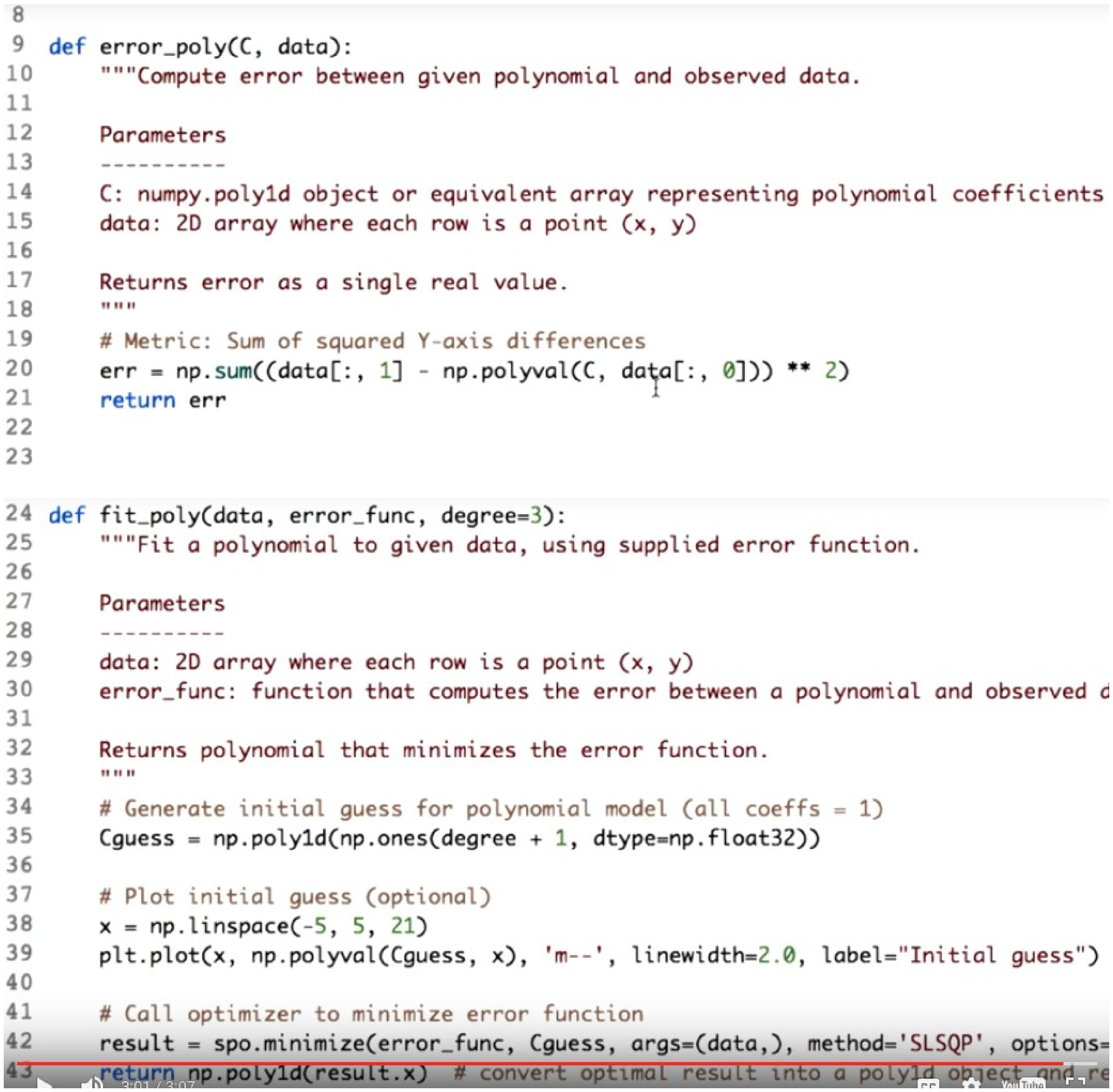


11. Now we'll look at some example code that can fit a line to data that's given. Remember, we're using a optimizer to do this. And first thing we have to do is describe for the optimizer what is the function it's trying to minimize. So we'll call this function error, and it takes two parameters, line and data. Line is just two coefficients, C0 and C1. And data is just a list of data, of course. Well, we've got some nice comments here that explain it, but really, our error is expressed simply in this single equation. We have the value of the actual data at each point here, minus the estimate that the line we're currently looking at would give at that same point. So we use the 0 coefficient, and the 1 coefficient, times the x value of the data at that point. So we take those differences and square it, and that's our error function. And that's what we're trying to minimize. We've added some code to illustrate how to use the minimizer to find the equation of a line. We start with our original line that the minimizer doesn't know. So, it's our secret, [LAUGH] but we're testing it to see if the minimizer can discover the equation of this line. Here's the equation for our line. It's just a two element array. It'll have a slope of four so coefficient zero is 4. And Y intercept of 2. So coefficient one of two. Here we generate X and Y values. Again keep in mind our minimizer doesn't know these but we are just generating them so we can look at them and we are plotting them for looking at it later. We take that original line and we use from numpie the random function to add some noise to it. So at each point along the X-axis, where we have data, we add some noise. So, now we've got our original line, plus some noise, and we're going to challenge our minimizer to find the equation for that original line, even though there's noise. We wrote a separate function fit_line that takes the data and the error function we defined and finds the equation for that line. Here's fit_line it does that for us. Two parameters the data, remember this is noisy data that is approximately a line and in other words we took our original line and added noise to it. And the error function, or the function we're trying to minimize. We have some nice comments here that tell us what those are, but now we just follow the steps like we've talked about before. We start with an initial guess. Here our initial guess is a slope of zero, and a mean of the rest of the data as our y intercept. It could be anything really, but that is a reasonable guess. We plot the initial guess so we have something to look at, and I'll show you that later. But here really is the meat of the function. You've seen it before. So we call our minimize function with the error_func. In other words, this is the function we're

trying to minimize. Our initial guess, and this is a parameter you haven't seen before, but this is a way by which we can pass the data to our error function. This is the method that we're going to use. And finally although it goes off the end here. We'll set display to true, which will mean we'll get to see any information as it goes along. So that's it. I mean really the key here for this is, this minimize call right here and then it returns the result. So let's run it and see what we get. There was some additional code that I skipped over that generated this plot. You can look at that on your own, of course. Okay, let's take a look. Our original line is this blue line. Of course the minimizer doesn't know anything about that. These green dots are our noisy data where we just added noise values to the blue line. Now we're asking our optimizer, okay find the equation of a line that best fits this data. The metric you're trying to minimize is error. So we passed it in an initial guess here of this purple line and this data. So that's all it knows right now is this initial guess of a purple line and this data. And then it iterates and tries different slopes and different y intercepts. Until finally, it converges to this red line and that's the solution. And I think it looks pretty decent. We can check it here, so if you look in the code, you'll see that our real line had a slope of four and a y intercept of 0.5. So we've got 4.17 and 0.64, not exactly. But if you look at the data you can see that it's pretty hard to know exactly what the underlying line would look like. So, I think our equation solver did a pretty good job.



```
Original polynomial:
     4      3     2
1.5 x - 10 x - 5 x + 60 x + 50
Optimization terminated successfully.    (Exit mode 0)
            Current function value: 27329.1186106
            Iterations: 8
            Function evaluations: 74
            Gradient evaluations: 8
Fitted polynomial:
       4         3         2
1.625 x - 10.55 x - 7.031 x + 64.63 x + 51.95
```



```python

def error_poly(C, data):
    """Compute error between given polynomial and observed data.

    Parameters
    ----------
    C: numpy.poly1d object or equivalent array representing polynomial coefficients
    data: 2D array where each row is a point (x, y)

    Returns error as a single real value.
    """
    # Metric: Sum of squared Y-axis differences
    err = np.sum((data[:, 1] - np.polyval(C, data[:, 0])) ** 2)
    return err


def fit_poly(data, error_func, degree=3):
    """Fit a polynomial to given data, using supplied error function.

    Parameters
    ----------
    data: 2D array where each row is a point (x, y)
    error_func: function that computes the error between a polynomial and observed d

    Returns polynomial that minimizes the error function.
    """
    # Generate initial guess for polynomial model (all coeffs = 1)
    Cguess = np.poly1d(np.ones(degree + 1, dtype=np.float32))

    # Plot initial guess (optional)
    x = np.linspace(-5, 5, 21)
    plt.plot(x, np.polyval(Cguess, x), 'm--', linewidth=2.0, label="Initial guess")

    # Call optimizer to minimize error function
    result = spo.minimize(error_func, Cguess, args=(data,), method='SLSQP', options=
    return np.poly1d(result.x)  # convert optimal result into a poly1d object and re
```

12. We can fit even more complicated functions to data like this. I'm going to show you the code of how to do that in just a moment. But I wanted to start with sort of the result, and then go back into the code and show you how we did it. So our original polynomial is a blue line. It's under here. You can't quite see it. And the noisy data, or the green dots there. This purple line is our initial guess. And the fitted polynomial, the red line here, fits the original pretty closely. So let me show you a little more detail. Here is output from our program. This is our original polynomial. It printed in kind of a weird way. Our original polynomial is 1.5 x to the fourth, minus 10 X to the third minus 5 X squared and so on. Down here are the results of our optimization. So here's what we got instead of 1.5 for the fourth power, we got 1.6 Instead of -10 for the third power we've got -10.5 and so on. But overall, pretty close and as you can see by that chart I just showed you, you know, very nice fitting. Let's look now at how we do that in code. The code here for, a higher-degree polynomial is very similar to what we had for the line. Again, there's an error function we're trying to minimize. And we take in the coefficients

for the polynomial and the actual data. And our error function is computed here. Again it's a sum of the difference between the actual data and the polynomial value squared. We take the sum of all those values and that's our error. So again very similar to what we did for the line. Here's our function that finds the coefficients of the polynomial has just a few parameters. The data the we're trying to fit our error function. In other words, how do we measure error and what are we trying to minimize? And the degree of the polynomial. We created an initial guess. In other words, what do we think the values of the coefficients are? And what we're doing here is we're just setting them all to be ones. We plot that, and then we call our minimizer, just like before. We have to tell it, what's the error function we're trying to minimize? What's our initial guess? We have to pass along the data, which then gets passed to the error function, and again, this method, SLS Q P and finally, you can't see it it is off to the side there, but same options essentially they are verbose options. And that's it that's how we use Python to create a model based on data.

13. Let's review what we learned. I showed you how to use a minimizer to find x such that f of x is minimized. I showed you how to minimize in multiple dimensions. And how to use a minimizer to build a parametrized model. Where can you go from here? There are a number of ways you can carry this forward. You can use functions besides polynomials, you can model stock prices, or you can optimize a portfolio.