# CSCI 416 - HW1

## Name:

In [ ]:
```
David Montenegro
```

## Problem 7

In [32]:
```
import numpy as np
A = np.array([[-2,-3],[1,0]])
B = np.array([[1,1],[1,0]])
x = np.array([[-1],[1]])
print ("A=\n", A)
print ("B=\n", B)
print ("x=\n", x)

C = np.linalg.inv(A)
print ("C=\n", C)

AC = np.matmul(A, C)
print ("AC=\n", AC)
CA = np.matmul(C, A)
print ("CA=\n", CA)

Ax = np.matmul(A, x)
print ("Ax=\n", Ax)

A_TA = np.matmul(A.T, A)
print ("A^TA=\n", A_TA)

AxMinusBx = np.matmul(A, x) - np.matmul(B, x)
print ("Ax-Bx=\n", AxMinusBx)

xNorm = np.dot(x.T, x)
xNorm = np.sqrt(xNorm[0,0])
print ("||x||=\n", xNorm)

AxMinusBxNorm = np.dot(AxMinusBx.T, AxMinusBx)
AxMinusBxNorm = np.sqrt(AxMinusBxNorm[0,0])
print ("||Ax-Bx||=\n", AxMinusBxNorm)

print ("first column of A=\n", A[:,[0]])

B[:,0:1] = x
print ("B=\n", B)

AColProduct = np.multiply(A[:,[0]],A[:,[1]])
print ("element-wise product between the first column of A and the second column of A=\n
```

```
A=
 [[-2 -3]
 [ 1  0]]
B=
 [[1 1]
 [1 0]]
x=
 [[-1]
 [ 1]]
```

```
C=
 [[ 0.          1.        ]
 [-0.33333333 -0.66666667]]
AC=
 [[1. 0.]
 [0. 1.]]
CA=
 [[1. 0.]
 [0. 1.]]
Ax=
 [[-1]
 [-1]]
A^TA=
 [[5 6]
 [6 9]]
Ax-Bx=
 [[-1]
 [ 0]]
||x||=
 1.4142135623730951
||Ax-Bx||=
 1.0
first column of A=
 [[-2]
 [ 1]]
B=
 [[-1  1]
 [ 1  0]]
element-wise product between the first column of A and the second column of A=
 [[6]
 [0]]
```

In [ ]:

```

```

In [ ]:

```

```

## Problem 8

*remember to complete part (1) to in hw1.pdf

## Imports

Run this code to import necessary modules. Note that the functions `cost_function` and `gradient` imported from module `gd` are stubs. You will need to fill in the code in `gd.py`.

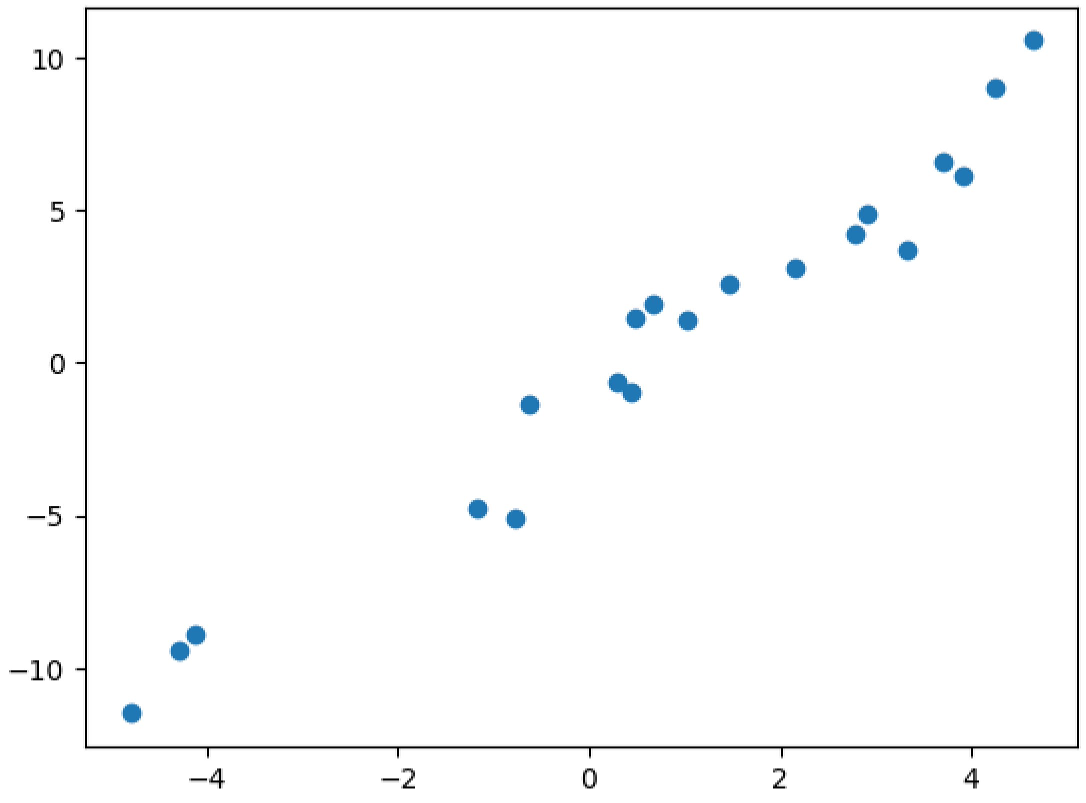
In [4]:

```
%matplotlib inline

%load_ext autoreload
%autoreload 2


import numpy as np
import matplotlib.pyplot as plt
from IPython.display import display, clear_output

from gd import cost_function, gradient  # stubs
```

## Create a simple data set

Run this cell to generate and plot some data from the linear model $y \approx -1 + 2x$, that is, $\theta_0 = -1$ and $\theta_1 = 2$.

In [5]:
```
# Set the random seed so the program will always generate the same data
np.random.seed(0)

# Generate n random x values between -5 and 5
n = 20
x = 10 * np.random.rand(n) - 5

# Generate y values from the model y ~= 2x - 1
epsilon = np.random.randn(n)
y = -1 + 2*x + epsilon

plt.plot(x, y, marker='o', linestyle='none')
plt.show()
```



## TODO: implement the cost function

The squared error cost function is

$$\frac{1}{2}\sum_{i=1}^{m}\left(h_\theta(x^{(i)}) - y^{(i)}\right)^2.$$

Open the file `gd.py` and implement `cost_function`. Then run this cell to test it out.

In [7]:
```
print(cost_function(x, y, 0,  1))    # should print 104.772951994
print(cost_function(x, y, 2, -1))    # should print 744.953822077
print(cost_function(x, y, -1, 2))    # should print 14.090816198
```

```
104.77295199433607
```

```
744.9538220768486
14.090816198013721
```

## Plotting setup

Run this cell. It sets up a routine `plot_model` that will be called later to illustrate the progress of gradient descent.

In [8]:
```
# Construct a dense grid of (theta_0, theta_1) values
theta0_vals = np.linspace(-10, 10)
theta1_vals = np.linspace(-10, 10)
[THETA0, THETA1] = np.meshgrid(theta0_vals, theta1_vals)

# Define a cost function that has x and y "baked in"
def mycost(theta0, theta1):
    return cost_function(x, y, theta0, theta1)

# Now vectorize this cost function and apply it simultaneously to all
# pairs in dense grid of (theta_0, theta_1) values
mycost_vectorized = np.vectorize(mycost)
J_SURF = mycost_vectorized(THETA0, THETA1)

# Define the test inputs
x_test = np.linspace(-5, 5, 100)

fig = plt.figure(1, figsize=(10,4))

# Create the figure
def init_plot():
    fig.clf();

    # Build left subplot (cost function)
    ax1 = fig.add_subplot(1, 2, 1);
    ax1.contour(THETA0, THETA1, J_SURF, 20)
    ax1.set_xlabel('Intercept theta_0')
    ax1.set_ylabel('Slope theta_1')
    ax1.set_xlim([-10, 10])
    ax1.set_ylim([-10, 10])

    # The data will be added later for these plot elements:
    line, = ax1.plot([], []);
    dot,  = ax1.plot([], [], marker='o');

    # Build right subplot (data + current hypothesis)
    ax2 = fig.add_subplot(1, 2, 2);
    ax2.plot(x, y, marker='o', linestyle='none')
    ax2.set_xlim([-6, 6])
    ax2.set_ylim([-10, 10])

    # The data will be added later for this:
    hyp, = ax2.plot( x_test, 0*x_test )

    return line, dot, hyp


# Define a function to update the plot
def update_plot(theta_0, theta_1, line, dot, hyp):
    line.set_xdata( np.append(line.get_xdata(), theta_0 ) )
    line.set_ydata( np.append(line.get_ydata(), theta_1 ) )
    dot.set_xdata([theta_0])
    dot.set_ydata([theta_1])
    hyp.set_ydata( theta_0 + theta_1 * x_test )
```

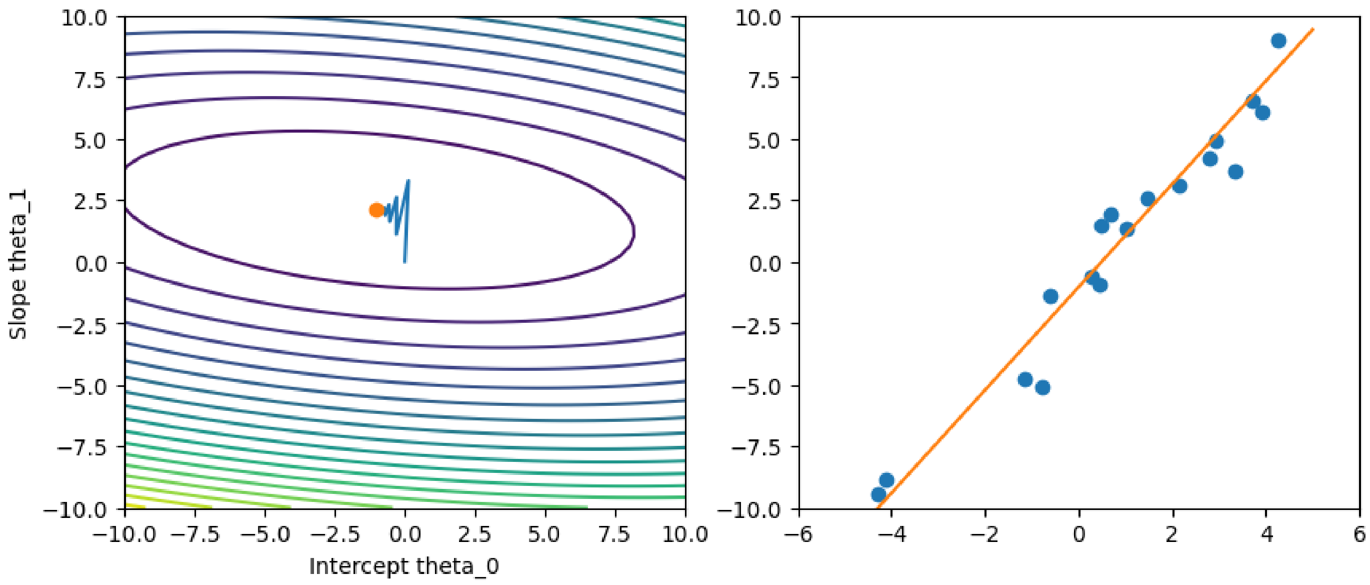
```
<Figure size 1000x400 with 0 Axes>
```

## TODO: implement gradient descent

In this cell you will implement gradient descent. Follow these steps:

1. Review the mathematical expressions for $\frac{\partial}{\partial \theta_0} J(\theta_0, \theta_1)$ and $\frac{\partial}{\partial \theta_1} J(\theta_0, \theta_1)$ for our model and cost funtion. (**Hint**: they are in the slides!)
2. Implement the function `gradient` in `gd.py` to return these two partial derivatives.
3. Complete the code below for gradient descent
   - Select a step size
   - Run for a fixed number of iterations (say, 20 or 200)
   - Update theta_0 and theta_1 using the partial derivatives
   - Record the value of the cost function attained in each iteration of gradient descent so you can examine its progress.

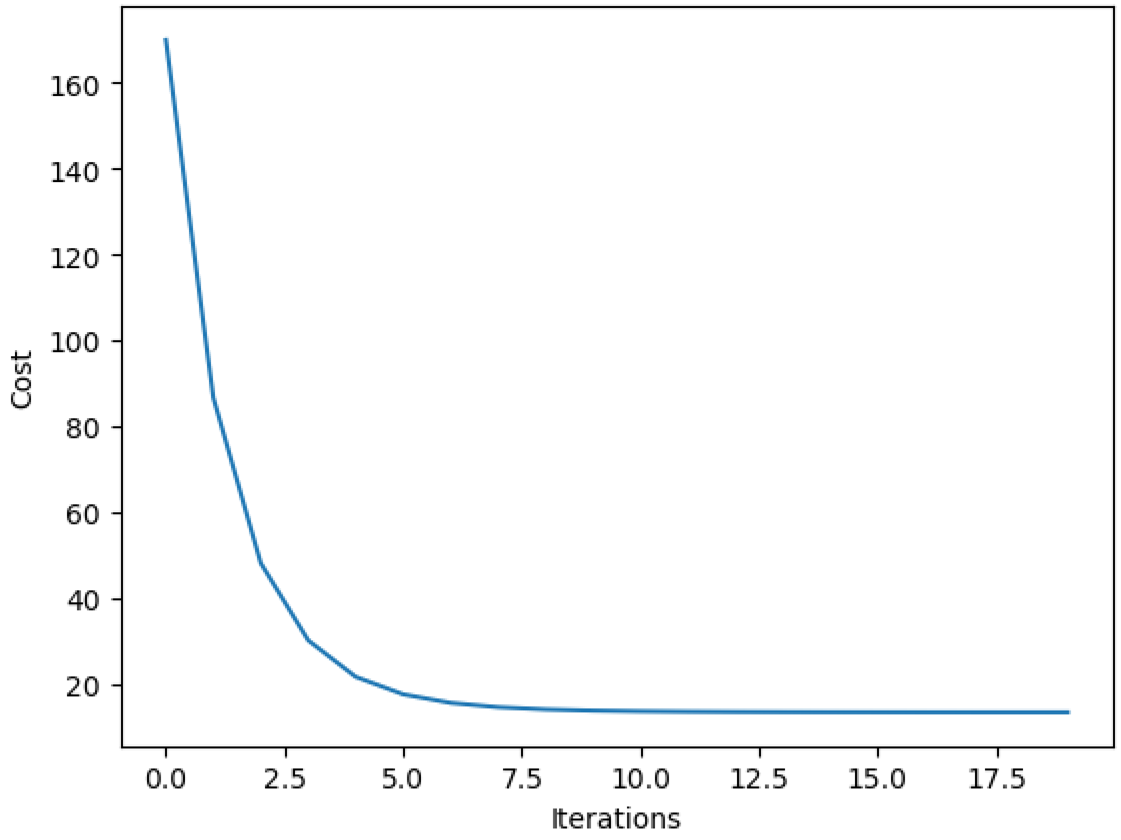
In [9]:
```
line, dot, hyp = init_plot()

iters = 20  # change as needed

# TODO: intialize theta_0, theta_1, and step size
theta_0 = 0
theta_1 = 0
step_size = 0.01
costs = np.zeros(iters)

for i in range(0, iters):

    # Uncomment the code below to display progress of the algorithm so far
    # as it runs.
    #
    clear_output(wait=True)
    update_plot(theta_0, theta_1, line, dot, hyp)
    display(fig)

    # TODO: write code to get partial derivatives (hint: call gradient in gd.py)
    # and update theta_0 and theta_1
    update_theta = gradient(x, y, theta_0, theta_1)
    theta_0 = theta_0-(step_size*update_theta[0])
    theta_1 = theta_1-(step_size*update_theta[1])
    costs[i] = cost_function(x, y, theta_0, theta_1)
```



## TODO: assess convergence

Plot the cost function vs. iteration. Did the algorithm converge? (Converging means it found the actual setting of $\theta$ that minimizes the cost. If the cost went up or did not go down as far as it could, it did not converge.)

```
In [10]: plt.plot(costs)
         plt.xlabel("Iterations")
         plt.ylabel("Cost")
         plt.show()
```



## TODO: experiment with step size

After you have completed the implementation, do some experiments with different numbers of iterations and step sizes to assess convergence of the algorithm. Report the following:

- A step size for which the algorithm converges to the minimum in at most 200 iterations
- A step size for which the algorithm converges, but it takes more than 200 iterations
- A step size for which the algorithm does not converge, no matter how many iterations are run

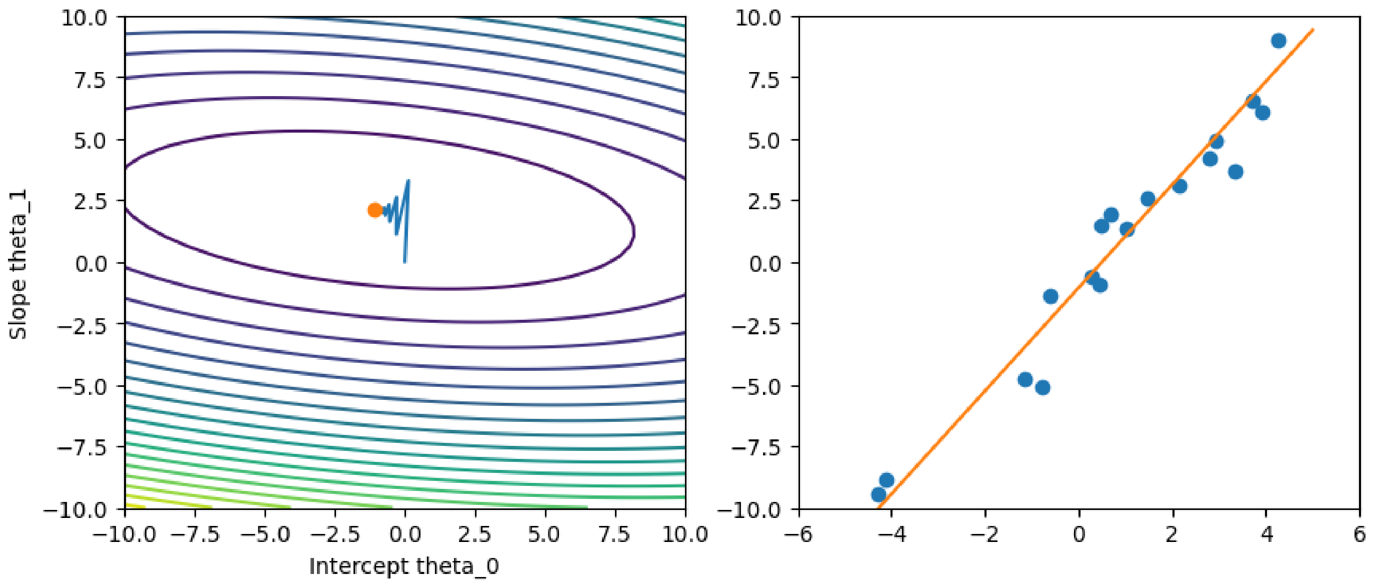
***Write your answer here*.**

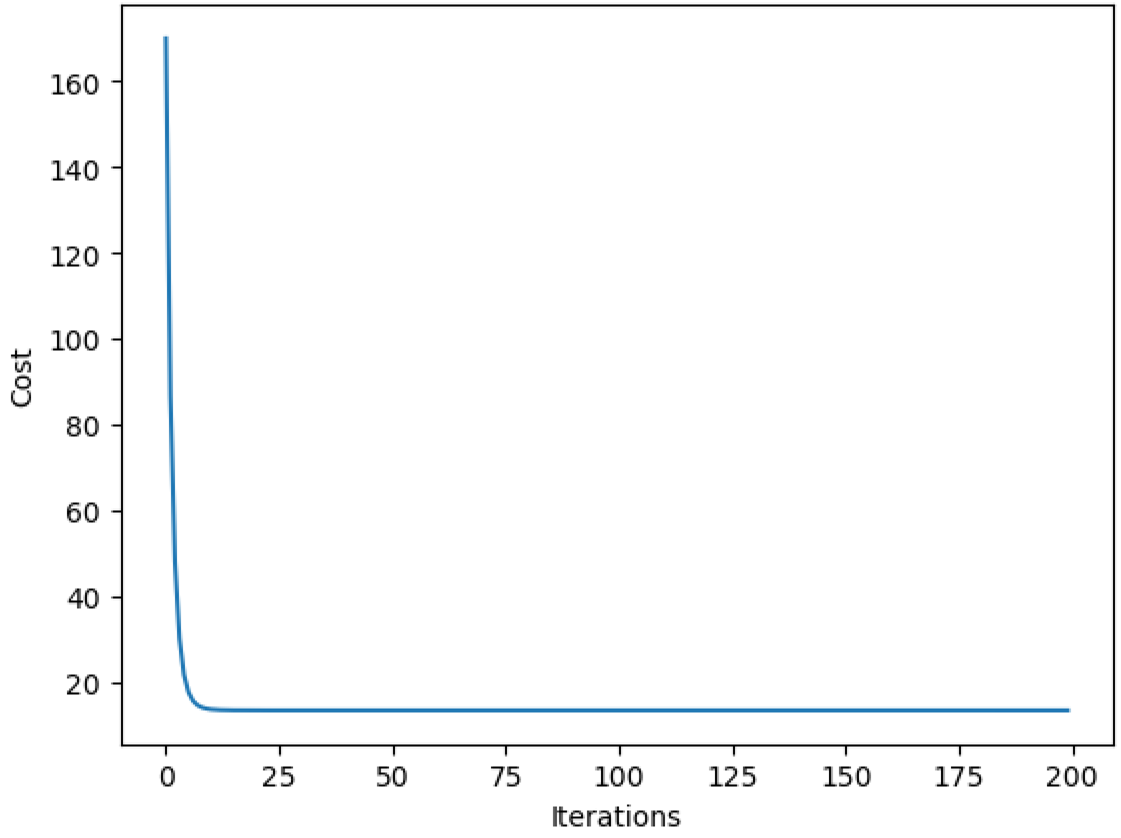
A step size for which the algorithm converges to the minimum in at most 200 iterations: 0.01

A step size for which the algorithm converges, but it takes more than 200 iterations: 0.0001

A step size for which the algorithm does not converge, no matter how many iterations are run: 0.1

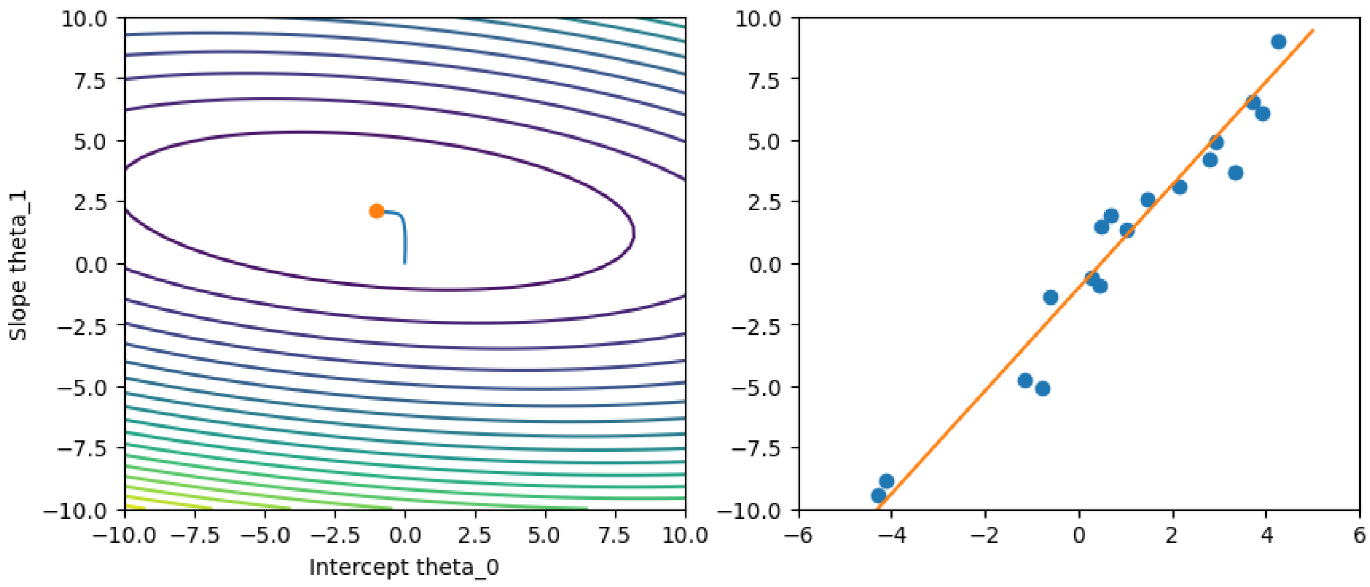
In [11]:
```
line, dot, hyp = init_plot()

iters = 200  # change as needed

# TODO: intialize theta_0, theta_1, and step size
theta_0 = 0
theta_1 = 0
step_size = 0.01
costs = np.zeros(iters)

for i in range(0, iters):

    # Uncomment the code below to display progress of the algorithm so far
    # as it runs.
    #
    clear_output(wait=True)
    update_plot(theta_0, theta_1, line, dot, hyp)
    display(fig)

    # TODO: write code to get partial derivatives (hint: call gradient in gd.py)
    # and update theta_0 and theta_1
    update_theta = gradient(x, y, theta_0, theta_1)
    theta_0 = theta_0-(step_size*update_theta[0])
    theta_1 = theta_1-(step_size*update_theta[1])
    costs[i] = cost_function(x, y, theta_0, theta_1)

plt.plot(costs)
plt.xlabel("Iterations")
plt.ylabel("Cost")
plt.show()

print ("theta0=\n", theta_0)
print ("theta1=\n", theta_1)
```

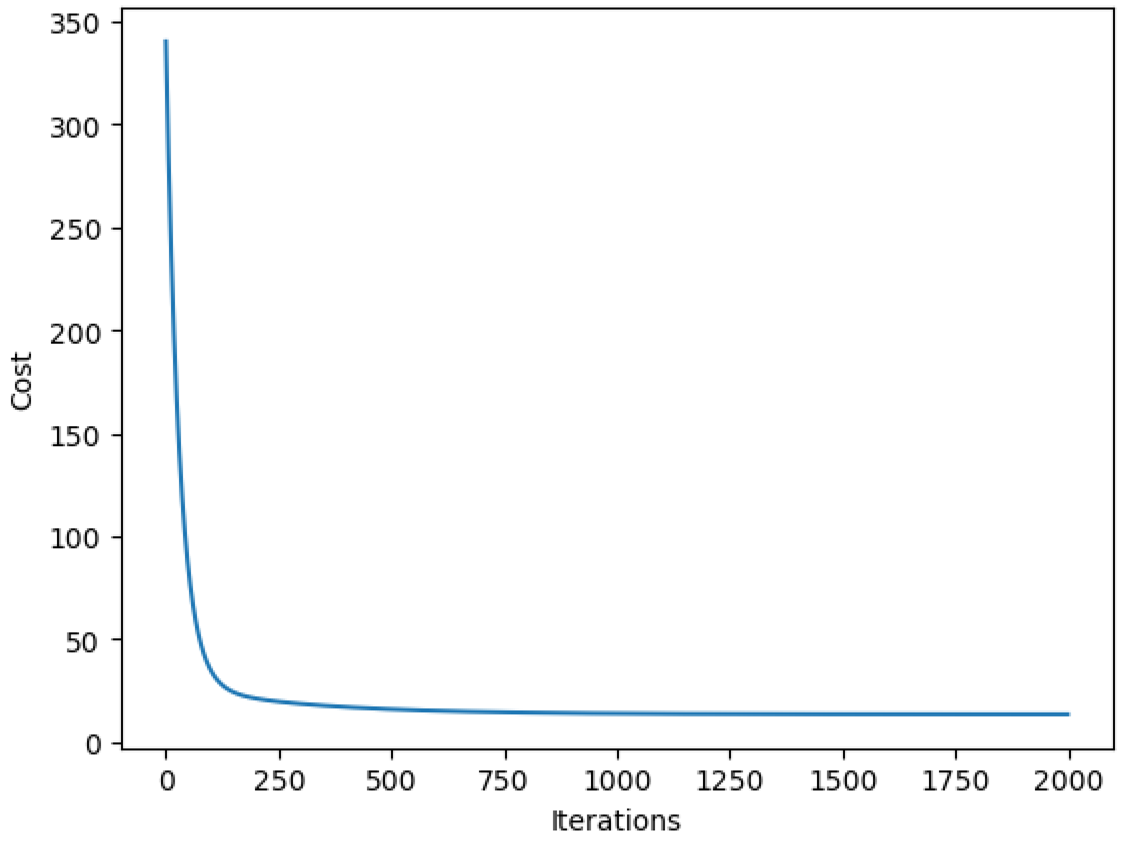




```
theta0=
 -1.0702535799250912
theta1=
 2.0934911262071134
```

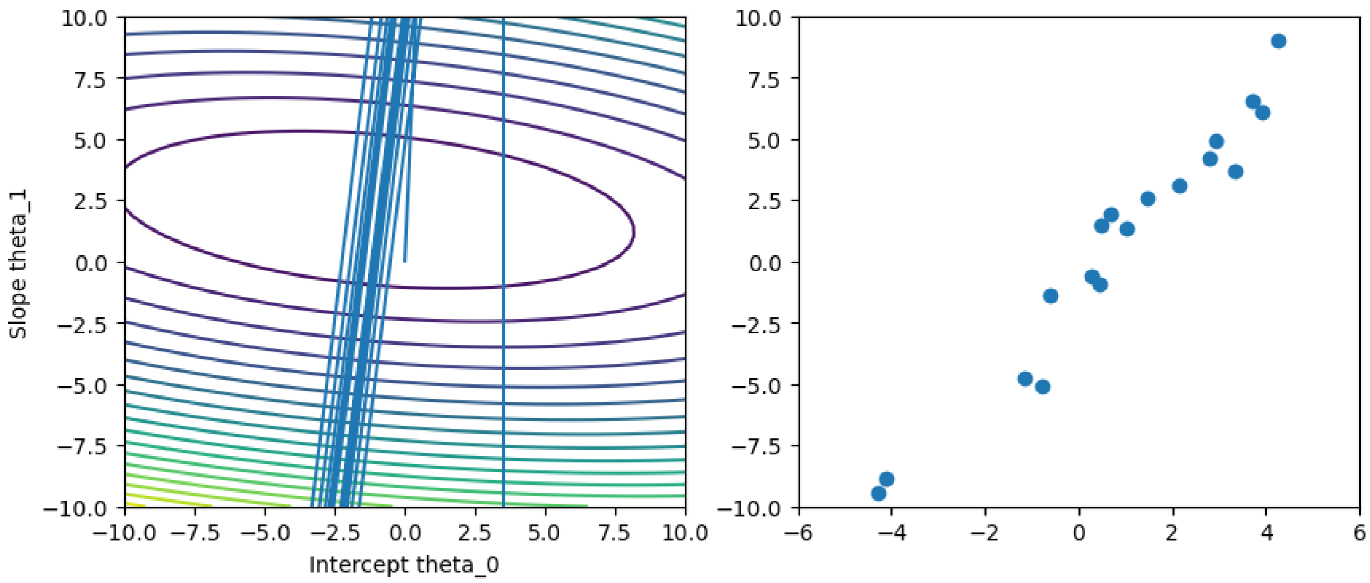
In [12]:
```
line, dot, hyp = init_plot()

iters = 2000  # change as needed

# TODO: intialize theta_0, theta_1, and step size
theta_0 = 0
theta_1 = 0
step_size = 0.0001
costs = np.zeros(iters)

for i in range(0, iters):

    # Uncomment the code below to display progress of the algorithm so far
    # as it runs.
    #
    clear_output(wait=True)
    update_plot(theta_0, theta_1, line, dot, hyp)
    display(fig)

    # TODO: write code to get partial derivatives (hint: call gradient in gd.py)
    # and update theta_0 and theta_1
    update_theta = gradient(x, y, theta_0, theta_1)
    theta_0 = theta_0-(step_size*update_theta[0])
    theta_1 = theta_1-(step_size*update_theta[1])
    costs[i] = cost_function(x, y, theta_0, theta_1)

plt.plot(costs)
plt.xlabel("Iterations")
plt.ylabel("Cost")
plt.show()
print ("theta0=\n", theta_0)
print ("theta1=\n", theta_1)
```

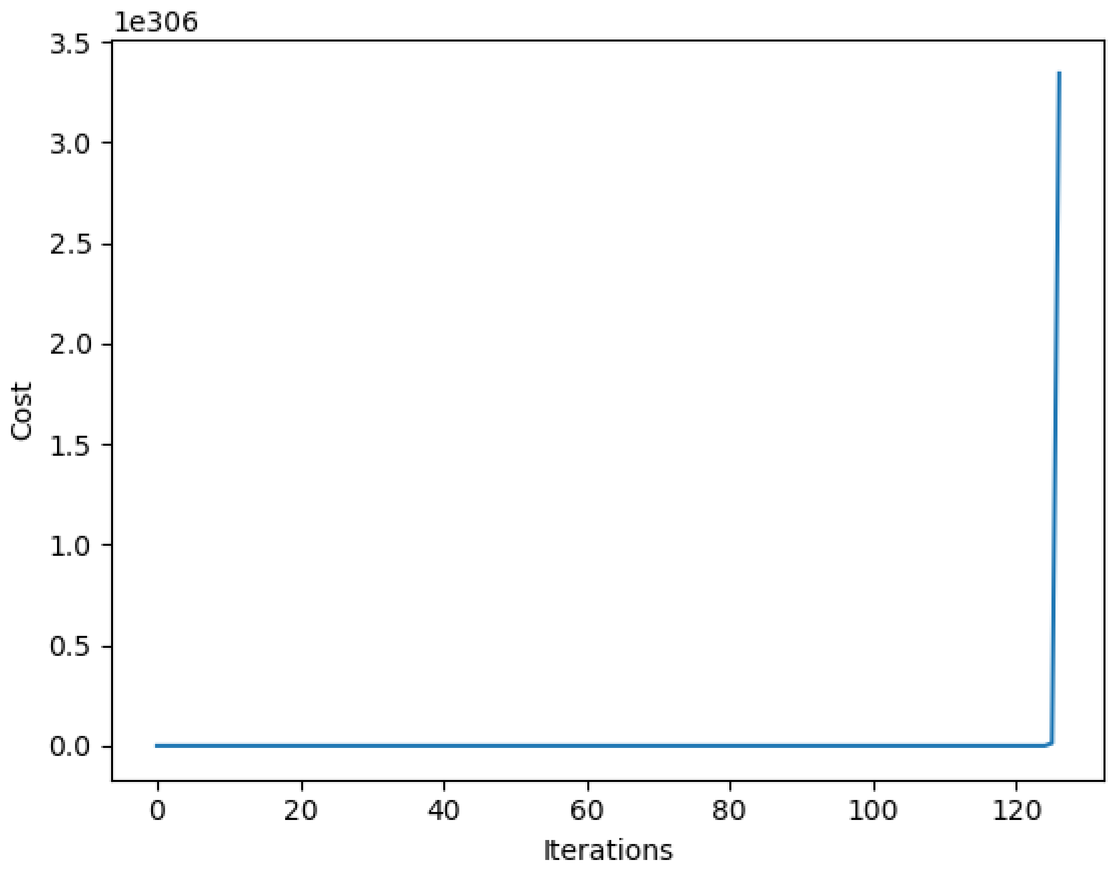




```
theta0=
 -1.0365632261935074
theta1=
 2.089762168167941
```

In [13]:
```
line, dot, hyp = init_plot()

iters = 500  # change as needed

# TODO: intialize theta_0, theta_1, and step size
theta_0 = 0
theta_1 = 0
step_size = 0.1
costs = np.zeros(iters)

for i in range(0, iters):

    # Uncomment the code below to display progress of the algorithm so far
    # as it runs.
    #
    clear_output(wait=True)
    update_plot(theta_0, theta_1, line, dot, hyp)
    display(fig)

    # TODO: write code to get partial derivatives (hint: call gradient in gd.py)
    # and update theta_0 and theta_1
    update_theta = gradient(x, y, theta_0, theta_1)
    theta_0 = theta_0-(step_size*update_theta[0])
    theta_1 = theta_1-(step_size*update_theta[1])
    costs[i] = cost_function(x, y, theta_0, theta_1)

plt.plot(costs)
plt.xlabel("Iterations")
plt.ylabel("Cost")
plt.show()
print ("theta0=\n", theta_0)
print ("theta1=\n", theta_1)
```





```
theta0=
 nan
theta1=
 nan
```

## Problem 9

## Problem Description

This notebook will guide you through implementation of **multivariate linear regression** to to solve the **polynomial regression** problem:

$$h_\theta(x) = \theta_0 + \theta_1 x + \theta_2 x^2 + \theta_3 x^3 + \theta_4 x^4 = \boldsymbol{\theta}^T \mathbf{x}$$

$$\boldsymbol{\theta} = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \theta_2 \\ \theta_3 \\ \theta_4 \end{bmatrix}, \qquad \mathbf{x} = \begin{bmatrix} 1 \\ x \\ x^2 \\ x^3 \\ x^4 \end{bmatrix}$$

Below, you will

1. Implement the cost function for multivarate linear regression
2. Implement the normal equations method to solve a multivariate linear regression problem
3. Implement gradient descent for multivariate linear regression
4. Experiment with feature normalization to improve the convergence of gradient descent

## Imports

Run this code.

```
In [14]: %matplotlib inline

import numpy as np
import matplotlib.pyplot as plt
```

## Helper functions

Run this code to set up the helper functions. The function `feature_expansion` accepts an vector of $n$ scalar x values and returns an $n \times 5$ data matrix by applying the feature expansion $x \mapsto [1, x, x^2, x^3, x^4]$ to each scalar $x$ value.

```
In [15]: def feature_expansion(x, deg):
    if x.ndim > 1:
        raise ValueError('x should be a 1-dimensional array')
    m = x.shape
    x_powers = [x**k for k in range(0,deg+1)]
    X = np.stack( x_powers, axis=1 )

    return X

def plot_model(X_test, theta):
    '''
    Note: uses globals x, y, x_test, which are assigned below
    when the dataset is created. Don't overwrite these variables.
    '''
    y_test = np.dot(X_test, theta)
    plt.scatter(x, y)
    plt.plot(x_test, y_test)
    plt.legend(['Test', 'Train'])
```

## (2 points) List comprehensions

Read about list comprehensions. Explain what is happening in the line of code

```
x_powers = [x**k for k in range(0,deg+1)]
```

***Your answer here***

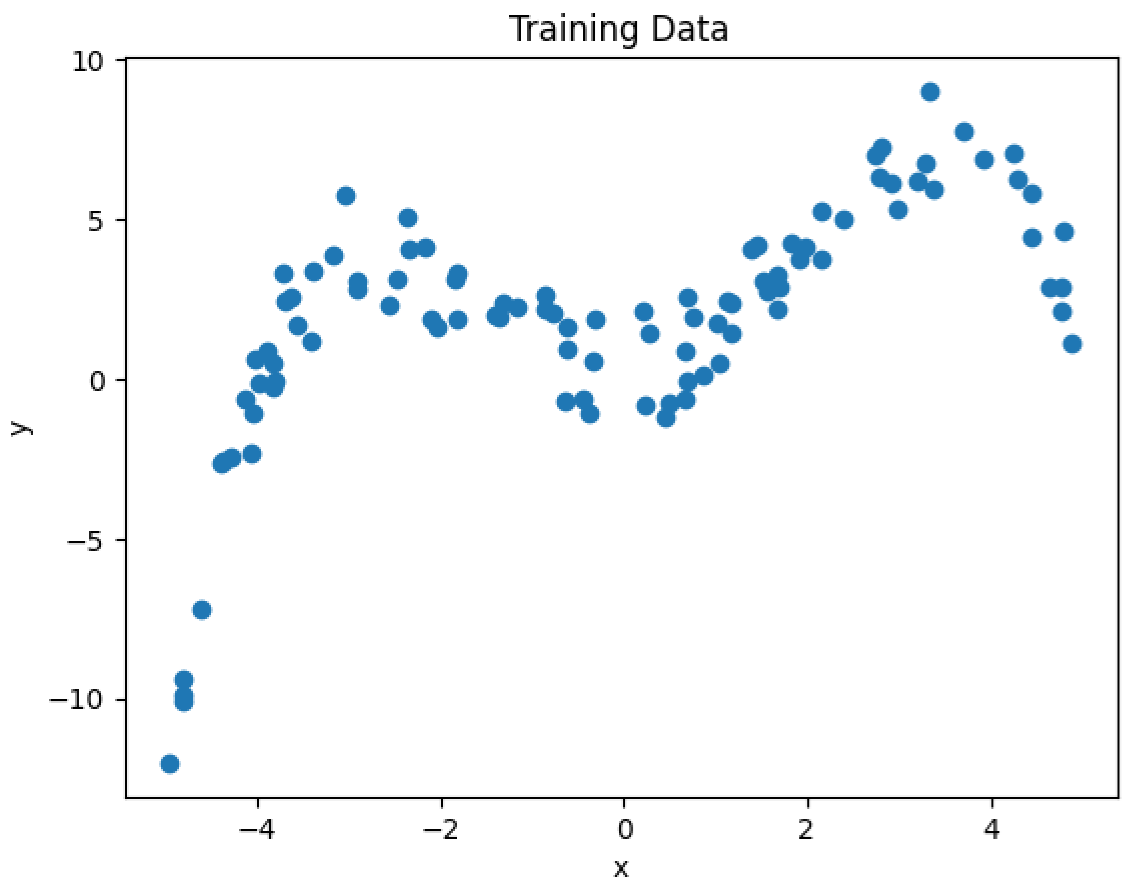
Here a new list is being initialized with the contents being the powers specified by the range being applied to each scalar x value. Instead of using a for loop, list comprehension can create the desired list faster.

## Create a data set for polynomial regression

Read and run the code below. This generates data from a fourth-degree polynomial and then uses feature expansion to set up the problem of learning the polynomial as multivariate linear regression

```
In [16]: # Set random seed
np.random.seed(0)

# Create random set of m training x values between -5 and 5
m = 100
```

```
x = np.random.rand(m)*10 - 5

# Create evenly spaced test x values (for plotting)
x_test  = np.linspace(-5, 5, 100)
m_test  = len(x_test);

# Feature expansion for training and test x values
deg = 4
X      = feature_expansion(x, deg)
X_test = feature_expansion(x_test, deg)

n = deg + 1   # total number of features including the '1' feature

# Define parameters (theta) and generate y values
theta = 0.1*np.array([1, 1, 10, 0.5, -0.5]);
y = np.dot(X, theta) + np.random.randn(m)   # polynomial plus noise

# Plot the training data
plt.scatter(x, y)
plt.title('Training Data')
plt.xlabel('x')
plt.ylabel('y')
plt.show()
```



## (2 points) Implement the cost function

Complete the code below to implement the cost function for multivariate linear regression.

In [17]:
```
def cost_function(X, y, theta):
    '''
    Compute the cost function for a particular data set and
    hypothesis (parameter vector)

    Inputs:
        X       m x n data matrix
        y       training output (length m vector)
        theta   parameters (length n vector)
    Output:
        cost    the value of the cost function (scalar)
    '''

    # TODO: write correct code to compute the cost function
    cost = 0
    sub = np.square(np.matmul(X, theta)-y)
    cost = np.sum(sub)/2
    return cost
```

## Test the cost function

Run this to test your cost function.

In [19]:
```
np.random.seed(1)

theta_random = np.random.rand(n)
theta_zeros  = np.zeros(n)
theta_ones   = np.ones(n)

print( "Cost function (random): %.2f" % cost_function(X, y, theta_random))  # prints 545
print( "Cost function  (zeros): %.2f" % cost_function(X, y, theta_zeros))   # prints 845
print( "Cost function   (ones): %.2f" % cost_function(X, y, theta_ones))    # prints 252
```

```
Cost function (random): 54523.64
Cost function  (zeros): 845.65
Cost function   (ones): 2524681.08
```

## (6 points) Implement first training algorithm: normal equations

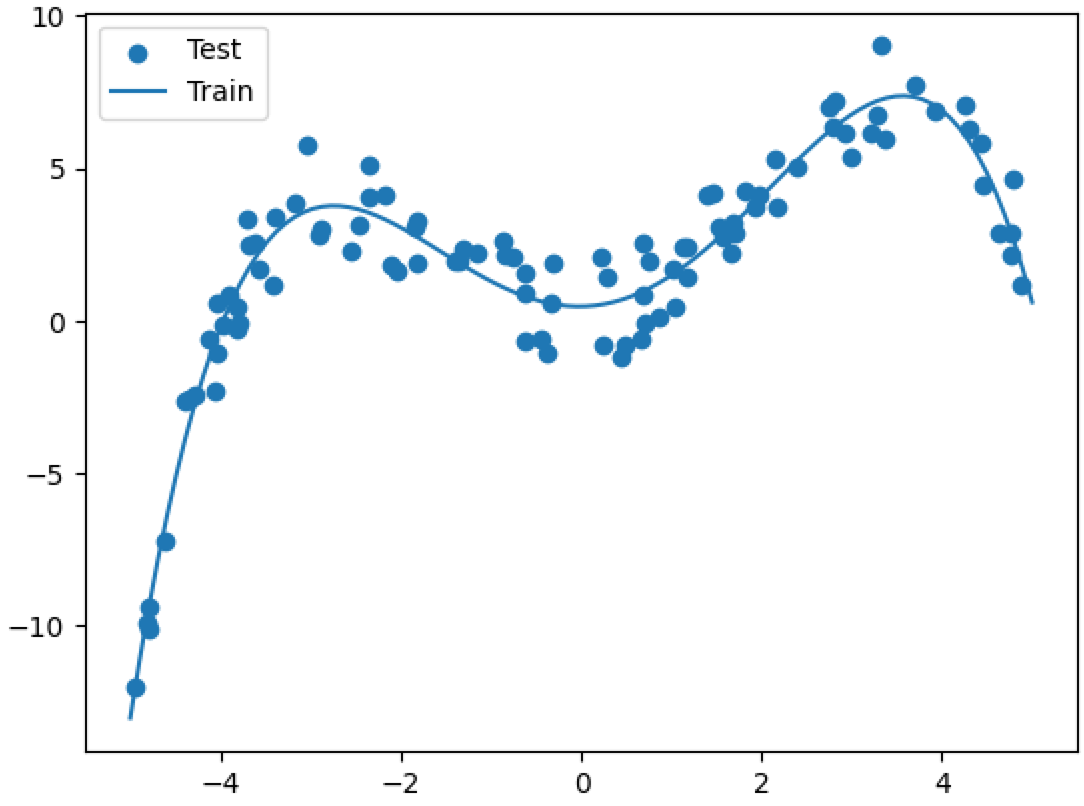
In [20]:
```
def normal_equations(X, y):
    '''
    Train a linear regression model using the normal equations

    Inputs:
        X       m x n data matrix
        y       training output (length m vector)
    Output:
        theta   parameters (length n vector)

    '''
    # TODO: write correct code to find theta using the normal equations
    m, n = X.shape
    theta = np.zeros(n)
    X_T = X.T
    X_TX = np.matmul(X_T, X)
    X_Ty = np.matmul(X_T, y)
    inverseX_TX = np.linalg.inv(X_TX)
    theta = np.matmul(inverseX_TX, X_Ty)
    return theta
```

## Use normal equations to fit the model

Run this code to test your implementation of the normal equations. If it runs properly you will see a curve that fits the data well. Note the value of the cost function for `theta_normal_equations`.

In [22]:
```
theta_normal_equations = normal_equations(X, y)
plot_model(X_test, theta_normal_equations)
print ("Cost function: %.2f" % cost_function(X, y, theta_normal_equations))
```

```
Cost function: 48.54
```



## (6 points) Implement second training algorithm: (vectorized) gradient descent

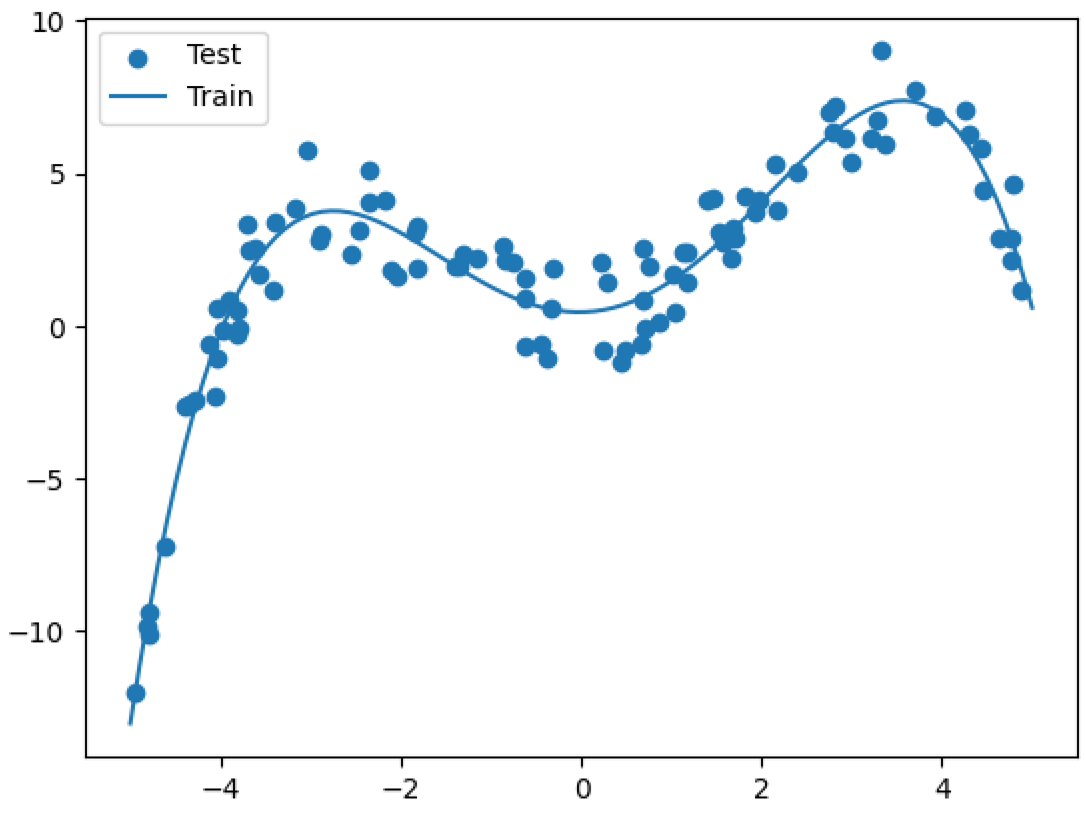
Implement gradient descent for multivariate linear regression. Make sure your solution is vectorized.

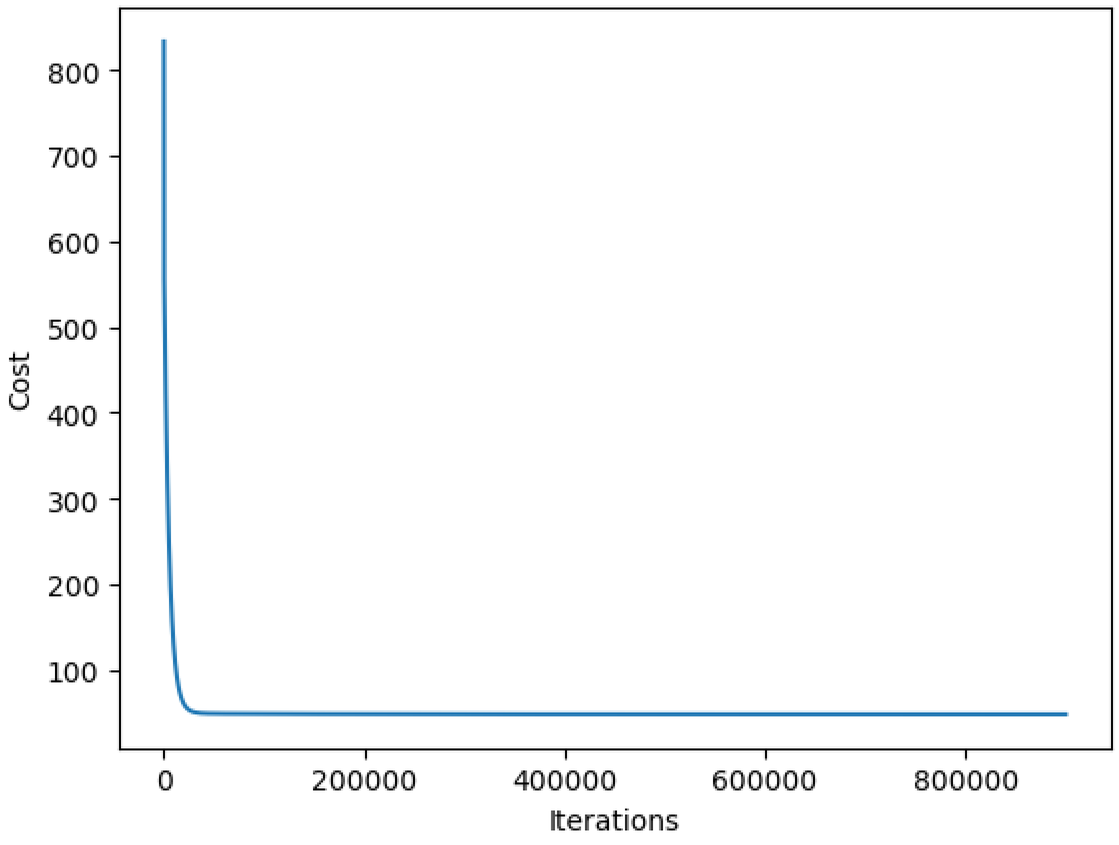
In [23]:
```
def gradient_descent( X, y, alpha, iters, theta=None ):
    '''
    Train a linear regression model by gradient descent

    Inputs:
        X       m x n data matrix
        y       training output (length m vector)
        alpha   step size
        iters   number of iterations
        theta   initial parameter values (length n vector; optional)

    Output:
        theta      learned parameters (length n vector)
        J_history  trace of cost function value in each iteration

    '''

    m,n = X.shape

    if theta is None:
        theta = np.zeros(n)

    # For recording cost function value during gradient descent
    J_history = np.zeros(iters)

    for i in range(0, iters):
        # TODO: compute gradient (vectorized) and update theta
        theta = theta - alpha*(1/m)*np.matmul(X.T, (np.matmul(X, theta) - y))
        # Record cost function
        J_history[i] = cost_function(X, y, theta)

    return theta, J_history
```

## (4 points) Use gradient descent to train the model

- Write code to call your `gradient_descent` method to learn parameter
- Plot the model fit (use `plot_model`)
- Plot the cost function vs. iteration to help assess convergence
- Print the final value of the cost function
- Experiment with different step sizes and numbers of iterations until you can find a good hypothesis. Try to match the cost function value from `normal_equations` to two decimal places. How many iterations does this take?

In [24]:
```
# TODO: write code
output = gradient_descent(X, y, 0.00001, 900000)
theta = output[0]
J_history = output[1]
plot_model(X_test, theta)
plt.figure(2)
plt.plot(J_history)
plt.xlabel("Iterations")
plt.ylabel("Cost")
plt.show()
print ("Cost function: %.2f" % J_history[-1])
print ("After 900000 iterations")
```





```
Cost function: 48.54
After 900000 iterations
```

## (10 points) Gradient descent with feature normalization

You should have observed that it takes many iterations of gradient descent to match the cost function value achieved by the normal equations. Now you will implement feature normalization to improve the convergence of gradient descent. Remember that the formula for feature normalization is:

$$x_j^{(i)} \leftarrow \frac{x_j^{(i)} - \mu_j}{\sigma_j}$$

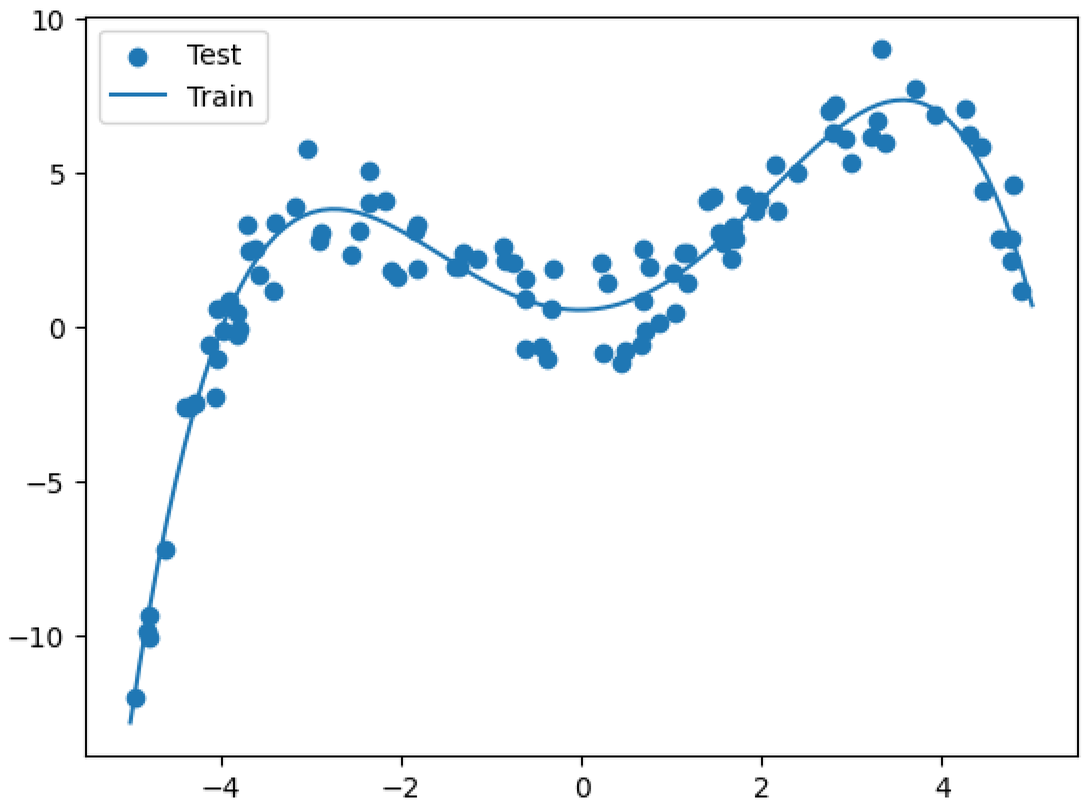
Here are some guidelines for the implementation:

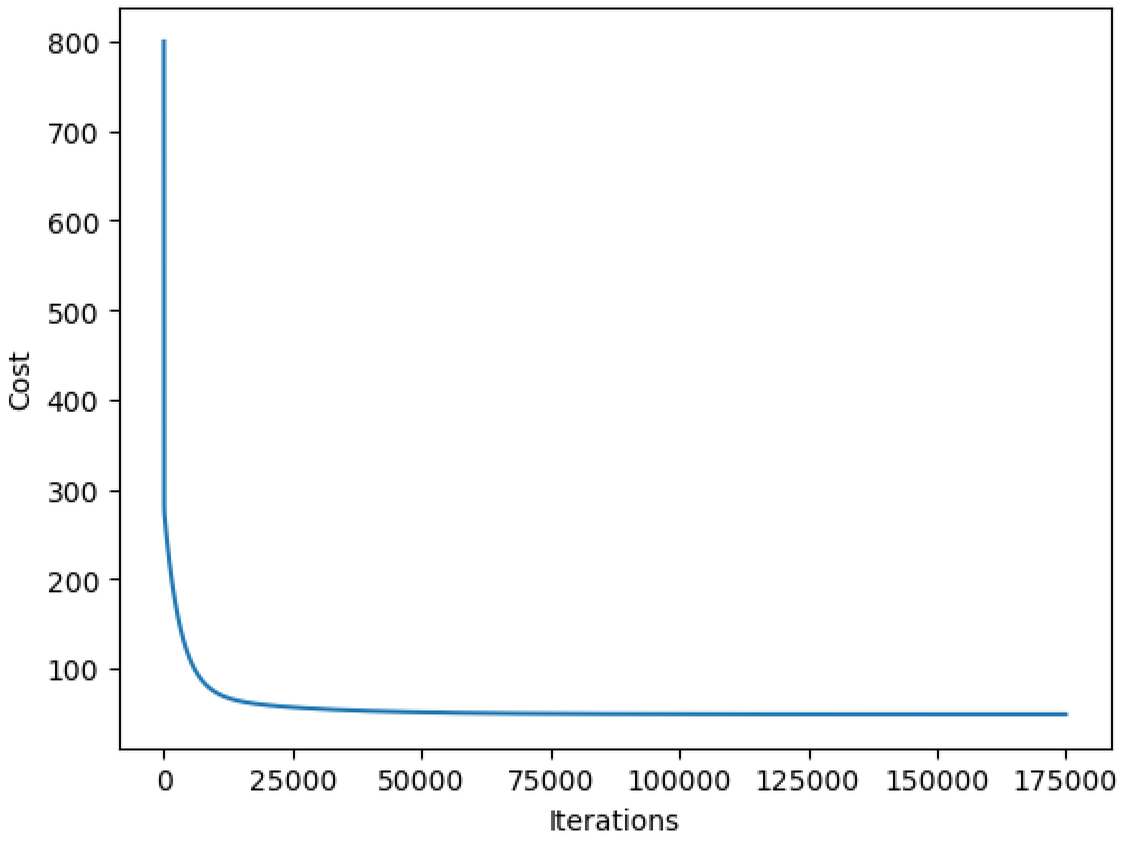
- The same transformation should be applied to train and test data.
- The values $\mu_j$ and $\sigma_j$ are the mean and standard deviation of the $j$th column (i.e., feature) in the **training data**. (Hint: there are numpy functions to compute these.)
- Do not normalize the column of all ones. (Optional question: why?)
- Use broadcasting to do the normalization--don't write for loops

After normalizing both the training data and test data, follow the same steps as above to experiment with gradient descent using the *normalized* training and test data: print the value of the cost function, and create the same plots. Tune the step size and number of iterations again to make gradient descent converge as quickly as possible. How many iterations does it take to match the cost function value from `normal_equations` to two decimal places?

In [25]:
```
# TODO: your code for gradient descent with feature normalization
XMean = np.mean(X)
```

```
XStd = np.std(X)
XNorm = (X-XMean)/XStd
X_testMean = np.mean(X_test)
X_testStd = np.std(X_test)
X_testNorm = (X_test-X_testMean)/X_testStd
output = gradient_descent(XNorm, y, 0.4, 175000)
theta = output[0]
J_history = output[1]
plot_model(X_testNorm, theta)
plt.figure(2)
plt.plot(J_history)
plt.xlabel("Iterations")
plt.ylabel("Cost")
plt.show()
print ("Cost function: %.2f" % J_history[-1])
print ("After 175000 iterations")
```





```
Cost function: 48.54
After 175000 iterations
```

***Write answer here: how many iterations?***

175000 iterations