Imports

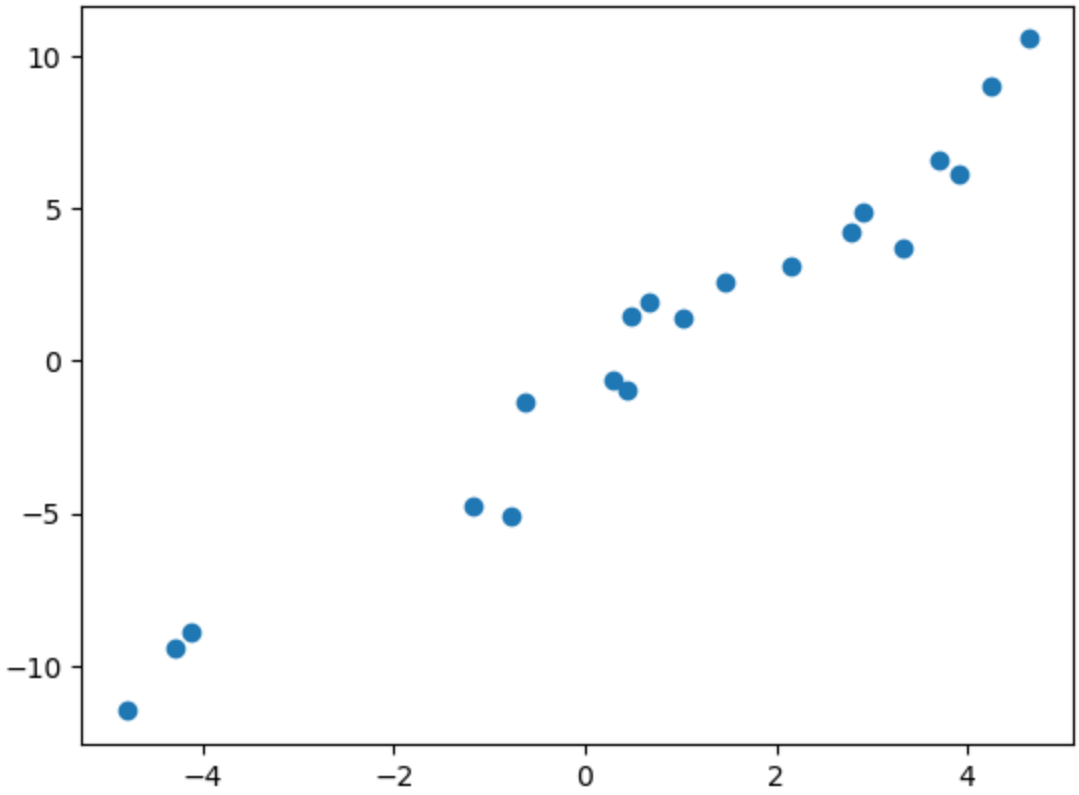
Run this code to import necessary modules. Note that the functions `cost_function` and `gradient` imported from module `gd` are stubs. You will need to fill in the code in `gd.py`.

In [1]:
```
%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 [2]:
```
# 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 [3]:
```
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
744.9538220768487
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 [4]:
```
# 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 )
```

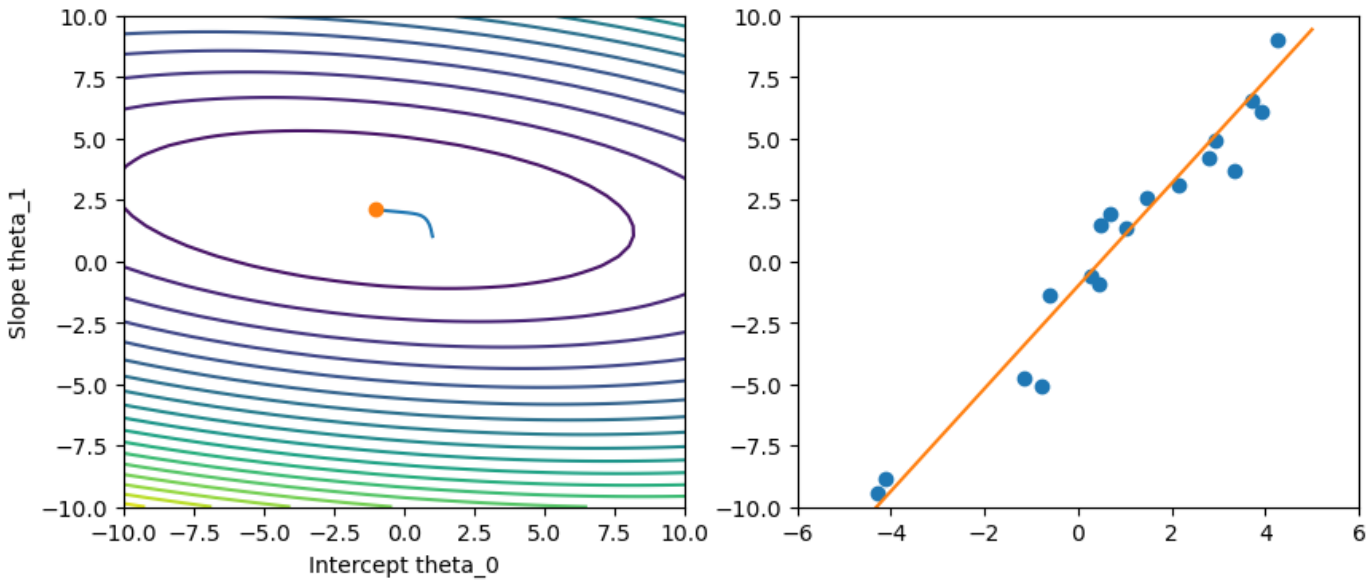
```
<Figure size 1000x400 with 0 Axes>
```

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 [47]:
```
line, dot, hyp = init_plot()

iters = 200  # change as needed

iteration = np.array([i for i in range(0, iters)])
costs = np.array([])
# TODO: intialize theta_0, theta_1, and step size

theta_0 = 1

theta_1 = 1

step = 0.001

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
    theta_0 = theta_0 - step * gradient(x, y, theta_0, theta_1)[0]
    theta_1 = theta_1 - step * gradient(x, y, theta_0, theta_1)[1]
    costs = np.append(costs, cost_function(x, y, theta_0, theta_1))
    pass
```

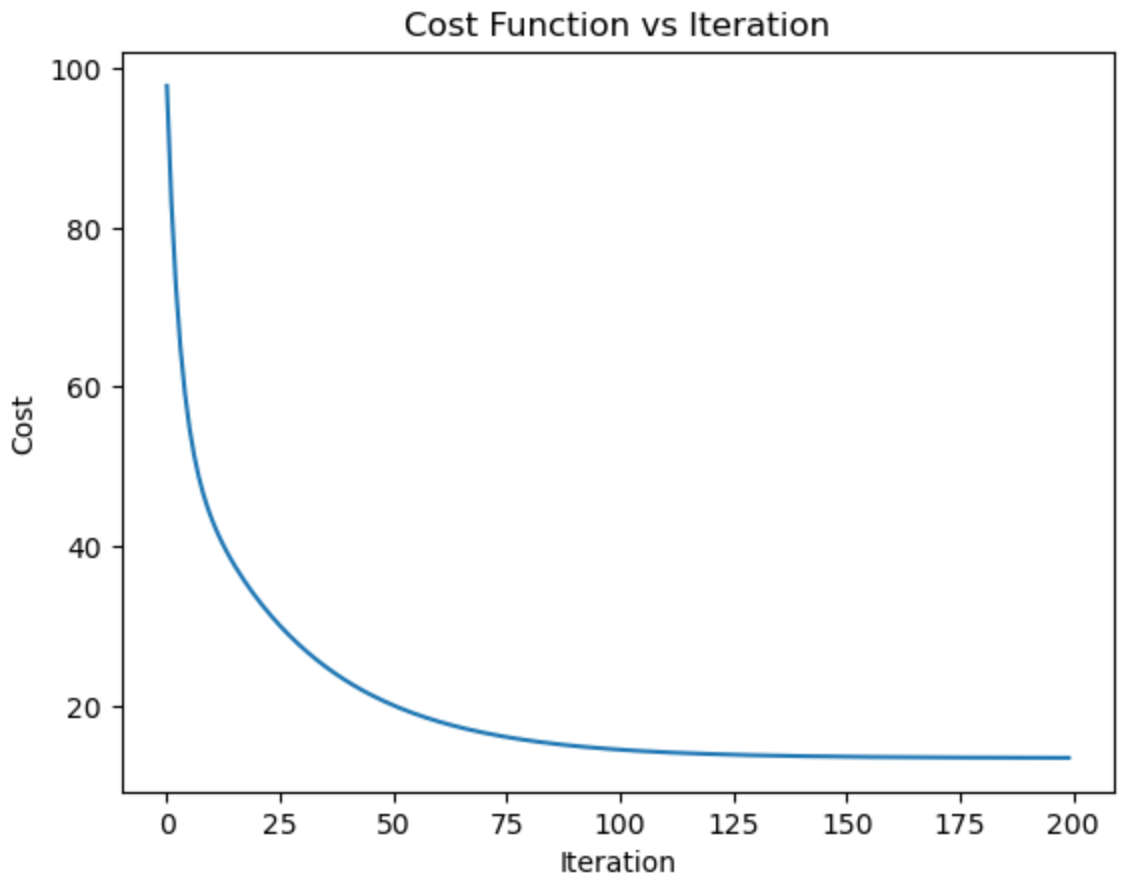


TODO: assess convergence

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

In [48]:
```
plt.plot(iteration,costs)
plt.title("Cost Function vs Iteration")
plt.xlabel("Iteration")
plt.ylabel("Cost")
```

Out[48]: Text(0, 0.5, 'Cost')



I would say that the algorithm did converge. Based on the plot above, it appears that the algortihm worked in decreasing the cost with each iteration, and each additional iteration, in turn, further decreased the cost.

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

A step size of 0.01 converges to the minimum at less than 200 iterations.

A step size of 0.0001 converges to the minimum at more than 200 iterations.

A step size of 0.1 does not converge.

In [ ]:

Describe your problem

Briefly describe what x and y are in your data, and why it may be interesting to predict y for a value of x that is not in the training set.

This data set compares the QBR of an NFL quarterback and the number of wins they had that season. While there are many factors that determine a wins team, the quarterback is arguably the most important position on the team, and their performance can greatly influence the teams performance. For teams looking to reach a certain number of wins, it would be helpful to know what QBR they should expect from their quarterback to give them the best chance of reaching that goal.

Enter or load your data

Either enter your x and y training data directly here as numpy arrays, or load the data from file. If you choose the latter, make sure to include your data file in the submission!

In [66]:
```
# Enter/load data here
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from gd import cost_function, gradient
from IPython.display import display, clear_output
from sklearn import datasets, linear_model
from sklearn.metrics import mean_squared_error, r2_score

#Pulling Data from Excel sheet and renaming columns
df = pd.read_excel('qbstats.xlsx')
df.columns = df.iloc[0]
df=df.drop(df.index[0])

#Putting QBR and Wins in numpy arrays
QBRS = np.array([df.iloc[:,2]])
wins = np.array([df.iloc[:,30]])
```

Fit a linear regression model

Fit a linear regression model to your data. You can either reuse the code from the problem you just completed, or you can use the linear regression model from scikit learn. Scikit learn is a large module of machine learning algorithms that we will be using throughout the course. It is included in the Anaconda distribution.

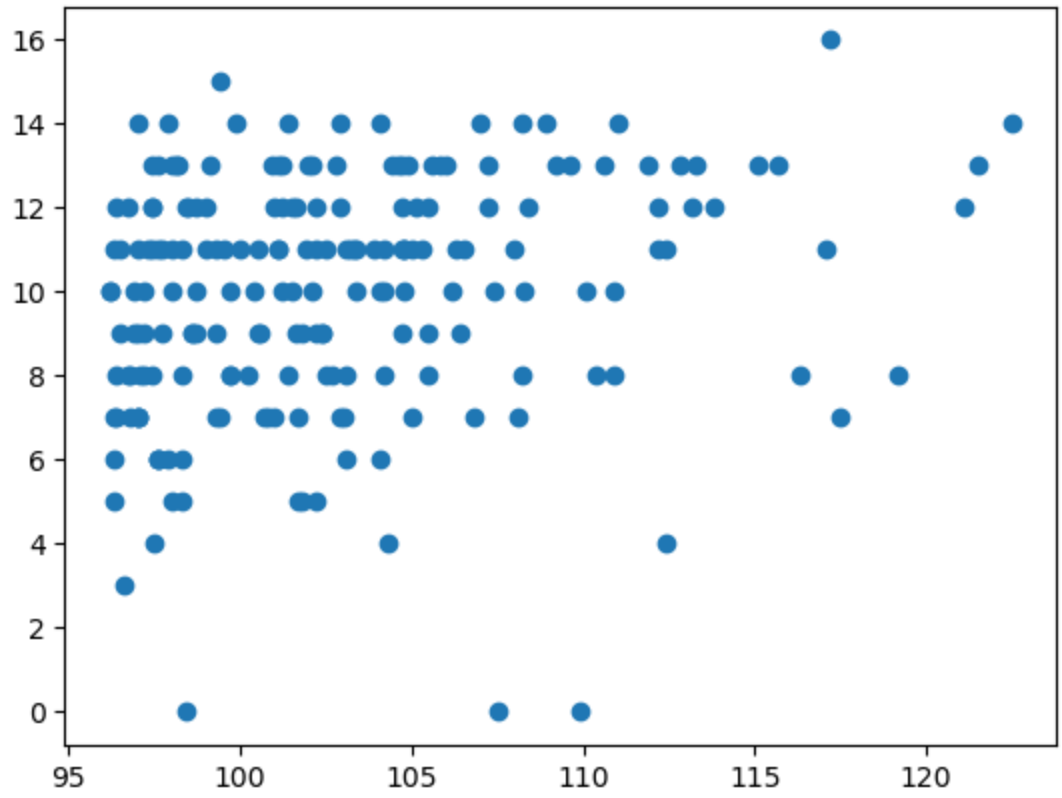
In [79]:
```
regr = linear_model.LinearRegression()

regr.fit(QBRS, wins)

wins_predict = regr.predict(QBRS)
```

Out[79]:
```
1
```

Plot the result

Plot your data and the best fitting hypothesis.

In [81]:
```
# Code to generate your plot here
plt.scatter(QBRS, wins)
plt.plot(QBRS, wins_predict, color = 'black', linewidth = 3)
plt.show()
```



Make a prediction

Use the learned hypothesis to make a prediction for an input value x that was not in the data set. Briefly discuss the result in the context of the data set you chose. Does the prediction seem useful?

In [ ]:
```
# Code to make the prediction
regr.fit(QBRS, wins)
QBR = np.array([108])
QBR = np.reshape(QBR,(-1,1))
wins_prediction = regr.predict(QBR)
```

This prediction sets a benchmark for how many games a team might expect to win based off of their quartback's performance. This would allow managers to determine whether or not their current quarterback provides their team with the best chance of reaching their goals.