In [1]:
```
from IPython import display
display.Image("../Data_cleaning_proj/IMG/bitcoin.jpg")
```

Out[1]:

Bitcoin is the first of it's kind in the cryptocurrency field. Back in 2009, an anonymous programmer that went by the pseudonym Satoshi Nakamoto sent out a newsletter with a code that would later become the backbone of Bitcoin's success. The program behind Bitcoin allows it to self-record/verify when a coin is spent, sent, etc., and it is in this way that Bitcoin is able to keep track of all the coins out in circulation. Rather than have a third party entity (such as a bank) accountable for auditing the circulation of Bitcoins it uses blockchains to monitor Bitcoin movements. There is a max cap of 21,000,000 Bitcoins (we are currently around 19MM) that will ever be created, so by Bitcoin being a) a high demand product, and b) limited in supply, is it able to appreciate in value at such an exponential rate.

Objective:

The intention of this study is to explore the rate of growth of Bitcoin, and to discover any features or metrics within in our dataset* that will help determine how to improve our accuracy for forecasting Bitcoin values and help evaluate if this is a promising investment.

Acknowledgements:

This post on linkedin helped to get started:

https://www.linkedin.com/pulse/blockchain-absolute-beginners-mohit-mamoria/

In [2]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
%matplotlib inline
import warnings
warnings.filterwarnings("ignore")
```

In [3]:
```
df = pd.read_csv('../Data_cleaning_proj/DATA/coin_Bitcoin_Final.csv',index_col = ['Date'], parse_dates=True)
df
```

Out[3]:

Date	open	high	low	close	volume	market_cap	Name	Symbol
2013-04-30	144.000000	146.929993	134.050003	139.000000	0.000000e+00	1.542813e+09	Bitcoin	BTC
2013-05-01	139.000000	139.889999	107.720001	116.989998	0.000000e+00	1.298955e+09	Bitcoin	BTC
2013-05-02	116.379997	125.599998	92.281898	105.209999	0.000000e+00	1.168517e+09	Bitcoin	BTC
2013-05-03	106.250000	108.127998	79.099998	97.750000	0.000000e+00	1.085995e+09	Bitcoin	BTC
2013-05-04	98.099998	115.000000	92.500000	112.500000	0.000000e+00	1.250317e+09	Bitcoin	BTC
...	...	...	...	...	...	...	...	...
2021-07-02	33549.600177	33939.588699	32770.680780	33897.048590	3.872897e+10	6.354508e+11	Bitcoin	BTC
2021-07-03	33854.421362	34909.259899	33402.696536	34668.548402	2.438396e+10	6.499397e+11	Bitcoin	BTC
2021-07-04	34665.564866	35937.567147	34396.477458	35287.779766	2.492431e+10	6.615748e+11	Bitcoin	BTC
2021-07-05	35284.344430	35284.344430	33213.661034	33746.002456	2.672155e+10	6.326962e+11	Bitcoin	BTC
2021-07-06	33723.509655	35038.536363	33599.916169	34235.193451	2.650126e+10	6.418992e+11	Bitcoin	BTC

2990 rows × 8 columns

The dataset consists of daily Bitcoin pricing information for the period of 04-29-2013 to 07-06-2021:

- High (highest price coin sold for the day)
- Low (lowest price coin sold for the day),
- Open (price a coin first trades with for that day),
- Closing (price a coin last trades with for that day),
- Volume (# of coins traded for that day),
- Marketcap (the total dollar-value of all the coins OR (price * total supply of bitcoins))

Cleaning the data:

In [4]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 2990 entries, 2013-04-30 to 2021-07-06
Data columns (total 8 columns):
 #   Column      Non-Null Count  Dtype  
---  ------      --------------  -----  
 0   open        2990 non-null   float64
 1   high        2990 non-null   float64
 2   low         2990 non-null   float64
 3   close       2990 non-null   float64
 4   volume      2990 non-null   float64
 5   market_cap  2990 non-null   float64
 6   Name        2990 non-null   object 
 7   Symbol      2990 non-null   object 
dtypes: float64(6), object(2)
memory usage: 210.2+ KB
```

In [5]:
```
df.shape
```

Out[5]:
```
(2990, 8)
```

In [6]:
```
df.isnull().sum()
```

Out[6]:
```
open          0
high          0
low           0
close         0
volume        0
market_cap    0
Name          0
Symbol        0
dtype: int64
```

To make it simpler to manage the dataframe as a time series object the dates column of the datset was converted into the index, thereby changing the type of the column into datetime.

Now we can also see from the above that dataset was already relatively clean before we started, as the rows are all uniform with 2,990 rows each and no null values appear to be in the data.

Descriptive Analysis:

we'll start to delve deeper into the data to see if there are insights we can gather by focusing on descriptive analysis for the closing prices

In [7]:
```
df.describe().T
```

Out[7]:

	count	mean	std	min	25%	50%	75%	max
open	2990.0	6.702342e+03	1.128929e+04	6.850500e+01	4.306560e+02	2.279110e+03	8.571748e+03	6.352375e+04
high	2990.0	6.895582e+03	1.164413e+04	7.456110e+01	4.366570e+02	2.392390e+03	8.735454e+03	6.486310e+04
low	2990.0	6.488134e+03	1.087023e+04	6.552600e+01	4.229225e+02	2.180860e+03	8.292635e+03	6.220896e+04
close	2990.0	6.713487e+03	1.129939e+04	6.843100e+01	4.309190e+02	2.295695e+03	8.577107e+03	6.350346e+04
volume	2990.0	1.090998e+10	1.889106e+10	0.000000e+00	3.048558e+07	9.480075e+08	1.594585e+10	3.509679e+11
market_cap	2990.0	1.209159e+11	2.109678e+11	7.784112e+08	6.309822e+09	3.755274e+10	1.500107e+11	1.186364e+12

a) Mean, Min, Max -

We can see from the chart that over the years the lowest price bitcoin sold on average on its closing day was 6,713, the lowest for 6.843, and the highest for 6.3503.

b) Standard Deviation -

The standard deviation (std) is an important metric here because of the frequent fluctuations Bitcoin prices experience. A low std would mean that the daily values stay close to the mean, while a high std would signify that daily prices are spread out. As expected we can see that the std for Bitcoin is 1.1299, meaning that Bitcoin prices could drop or appreciate in value by around 1.1000 on a daily basis, further confirming its high volatility.

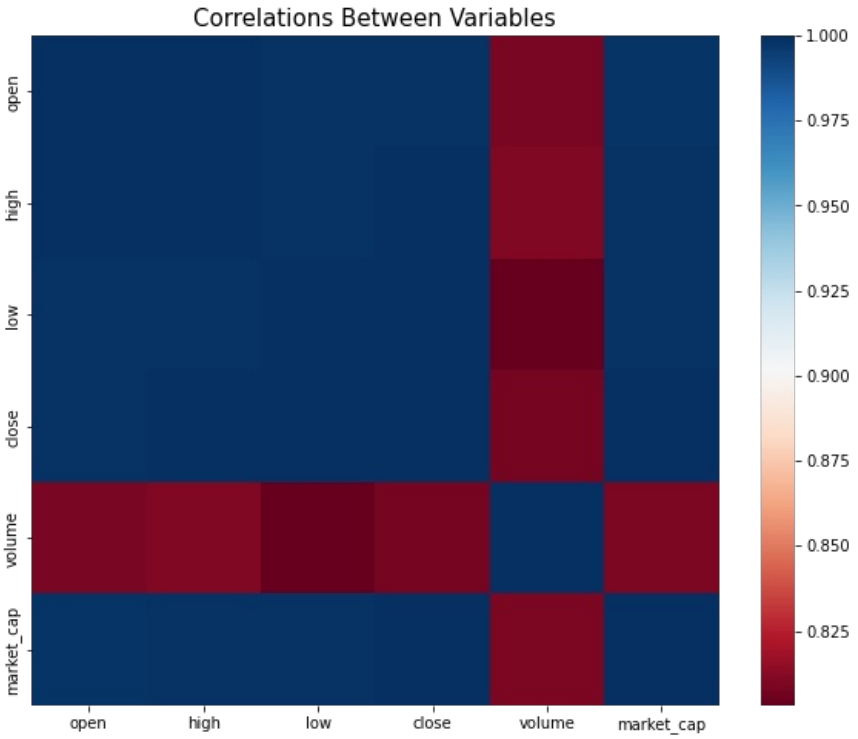
c) Percentiles -

Percentiles is the percentage of values that fall below a given value. For example, the 25th percentile of the close column shows that 25 percent of the close Bitcoin values fell below 4.3091, 50 percent of the prices were below 2.29569, and 75 percent of the prices were 8.577 or lower.

Visual Analysis:

Checking Correlations Between Variables

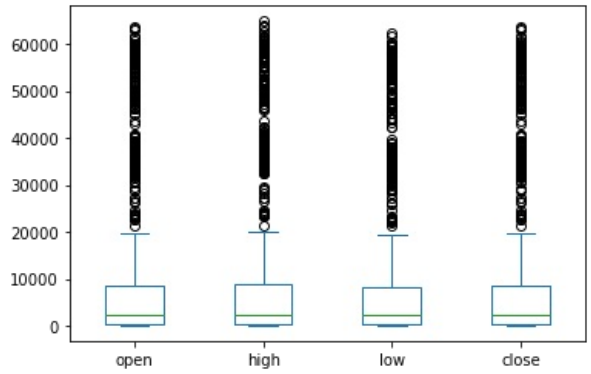
In [8]:
```
plt.figure(figsize=(10,8))
sns.heatmap(df.corr(), cmap="RdBu")
plt.title("Correlations Between Variables", size=15)
plt.show()
```



Ploting Box plot to see Extreme values

In [9]:
```
df = pd.DataFrame(df, columns=['open', 'high', 'low', 'close'])
df.plot.box()
```

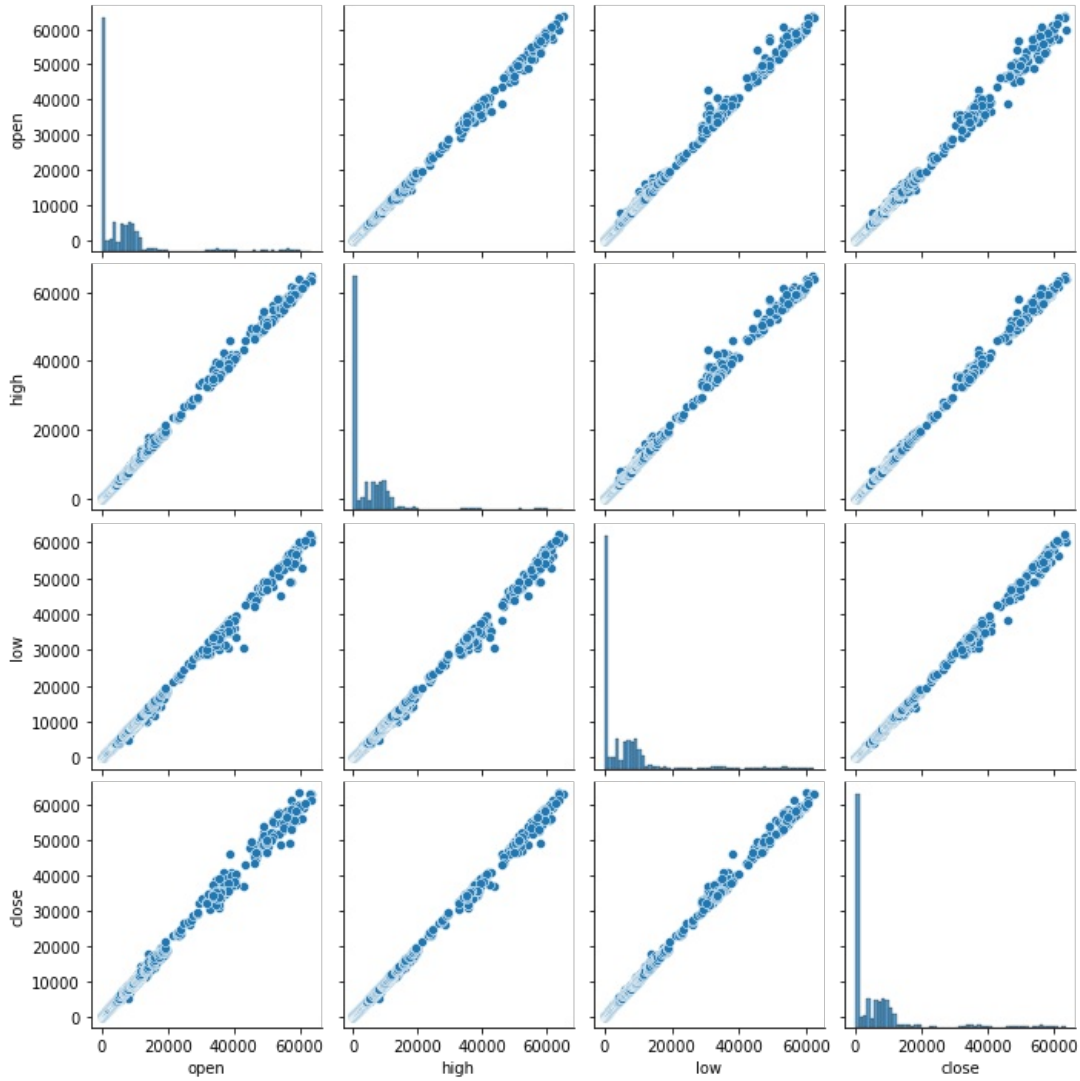
Out[9]:
```
<AxesSubplot:>
```



Visualizing the Correlation between the numerical variables using pairplot visualization

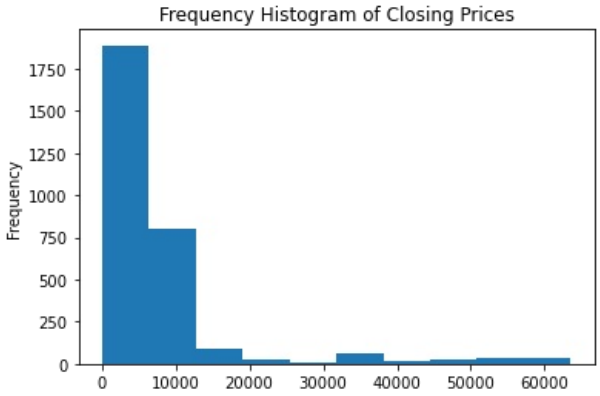
In [10]:
```
import matplotlib.pyplot as plt
import seaborn as sns
%matplotlib inline
sns.pairplot(data=df)
```

Out[10]:
```
<seaborn.axisgrid.PairGrid at 0x19246d9b700>
```



To visually illustrate our findings we can implement graphs such as a histogram to help see the distribution of our data, we can start this by first using the matplotlib module.

In [11]:
```
import matplotlib.pyplot as plt

plt.hist(df['close'], bins=10)
plt.gca().set(title='Frequency Histogram of Closing Prices', ylabel='Frequency')
plt.show()
```



The histogram displays the majority of Bitcoin prices to be lower than 20,000. In fact, 2,787 out of the total 2,990 points are below 20,000.00

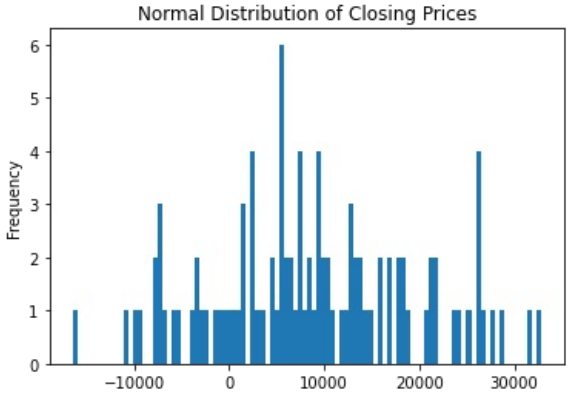
In [12]:
```
df[df['close'] < 20000].count()
```

Out[12]:
```
open     2787
high     2787
low      2787
close    2787
dtype: int64
```

Normal Data Distribution -

By creating a distribution chart we center our data around the mean as percentage of the total. A normal distribution would look like a bell curve, and from our graph below we can see most values fall near the mean but also it shows some skewness towards the right.

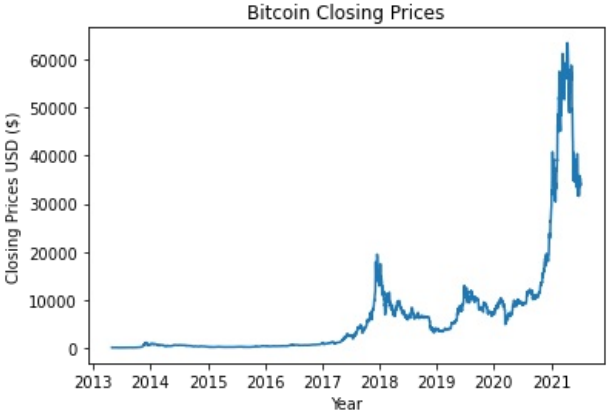
In [13]:
```
x = np.random.normal(df['close'].mean(), df['close'].std(), 100)
plt.gca().set(title='Normal Distribution of Closing Prices', ylabel='Frequency')
plt.hist(x, 100)
plt.show()
```



Line Plots -

A line plot places all points of our data into one graph making it easier to see any trends the data may be showing. The chart below shows that Bitcoin prices were primarily low in price up until 2018. The graph appears to indicate growth and versatility for Bitcoin occurred mostly after 2018. Otherwise, it seems that Bitcoin has had positive growth over the past decade.

In [14]:
```
plt.plot(df.index,df['close'])
plt.gca().set(title='Bitcoin Closing Prices', xlabel='Year', ylabel='Closing Prices USD ($)')
plt.show()
```



Visualization of closing price (on row data & log-scale)

Idea: The reason why log-scale is used on y-axis is that it reveals the percentile change

Ex:

Case 1) when price goes up from $ 10-$ 15: change(increase) is $5. Increase rate is 50%

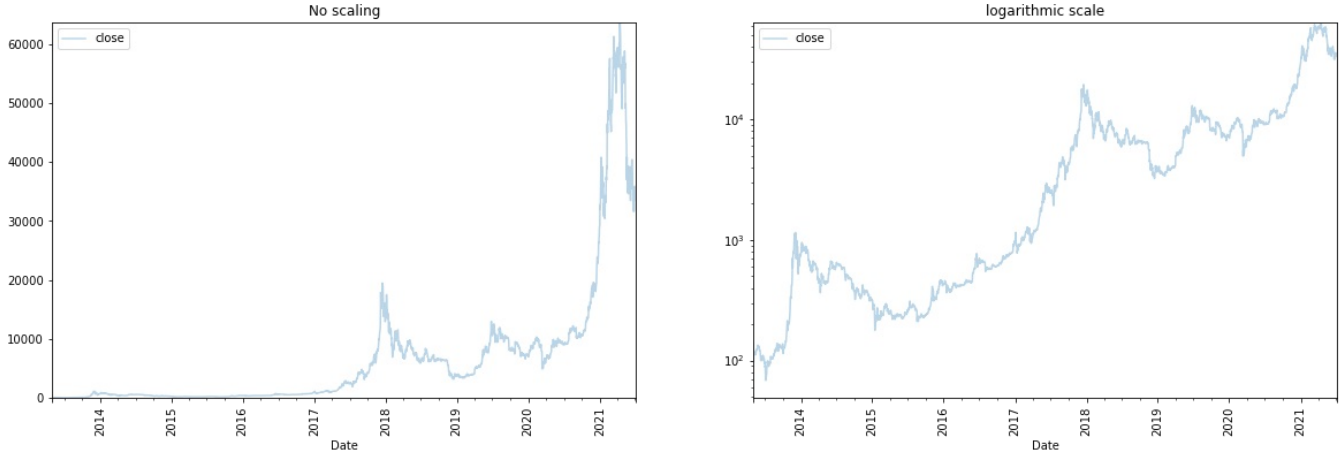
Case 2) when price goes up from $20-$25: change(increase) is $5. Increase rate is 25%

In both cases , change is same but rate of change is different.

Refrence: What is the difference between a logarithmic price scale and a linear one?
http://www.investopedia.com/ask/answers/05/logvslinear.asp#ixzz4pKMuY5HA

In [15]:
```
plt.figure(num=None, figsize=(20, 6))
plt.subplot(1,2,1)
ax = df['close'].plot(style=['-'])
ax.lines[0].set_alpha(0.3)
ax.set_ylim(0, np.max(df['close'] + 100))
plt.xticks(rotation=90)
plt.title("No scaling")
ax.legend()
plt.subplot(1,2,2)
ax = df['close'].plot(style=['-'])
ax.lines[0].set_alpha(0.3)
ax.set_yscale('log')
ax.set_ylim(0, np.max(df['close'] + 100))
plt.xticks(rotation=90)
plt.title("logarithmic scale")
ax.legend()
```

Out[15]:
```
<matplotlib.legend.Legend at 0x1924900cc10>
```



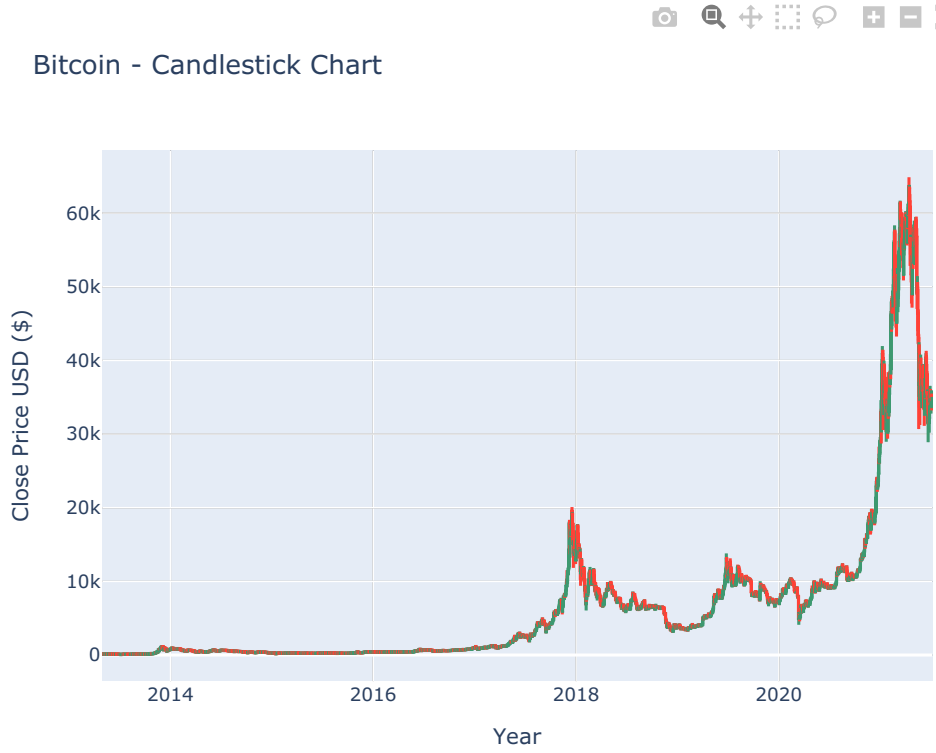
Some features of the plot above:

- There is an uppward trend from 2016 for each graph
- There is no seasnality
- There are no outliers

Candlestick Plot Analysis:

The benefits of a candlestick chart are the visibility they bring to trends. This candlestick simultaneously shows the market's open, high, low, and close price for the day. The red lines visualize a fall in prices and the green lines show days where there are an increase in prices. While there were many days where prices decreased they were paired with an equal amount of days with price increases. The greatest changes in price appear to occur after 2017 so we'll take a closer look at this period.

In [16]:
```
import plotly.graph_objects as go
figure = go.Figure(data=[go.Candlestick(x = df.index,
                                        open = df["open"],
                                        high = df["high"],
                                        low = df["low"],
                                        close = df["close"])])
figure.update_layout(title = "Bitcoin - Candlestick Chart", xaxis_title='Year',yaxis_title='Close Price USD ($)',
                     xaxis_rangeslider_visible = False)
figure.show()
```



Futher Examination: Closing Prices from 2017 - 2021

Closing prices from 2017 through the end of 2020 illustrate prices hovering between the range of 10,000 to 20,000. Interesting to note is that Bitcoin's prices increased and decreased the most during the year of 2021.

In [17]:
```
import plotly.express as px
date_start = '2017-07-01'
date_end = '2021-07-06'
figure = px.line(df, x = df.index,
                 y = 'close',
                 range_x = [date_start,date_end],
                 title = "Bitcoin Analysis {} to {}".format(date_start, date_end))
figure.show()
```

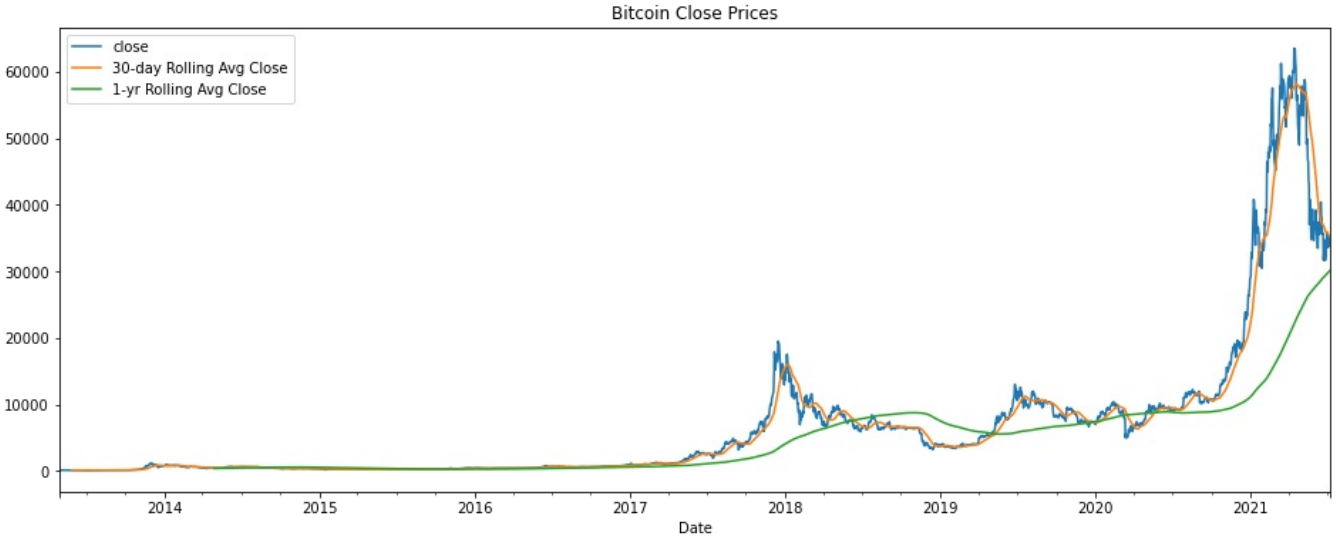
Rolling Averages:

Rolling averages is the average price of Bitcoin prices over a certain time period, calculated repeatedly. For example, a moving average can be calculated by adding prices from the most recent days (in the chart below it's, the last 30 days, and 365 days) and dividing by that

same number of days (or 10, and 365, respectively). Moving averages help smooth out erratic movement that may be present in Bitcoin's price on a given day. The below graph suggests that the 30-day and 1-year averages of closing prices overall follow the direction of the daily pricing, thus confirming again that Bitcoin prices are moving in a positive upward direction.

In [18]:
```
df['close'].plot(title='Bitcoin Close Prices',label='close',legend=True)
df.rolling(30).mean()['close'].plot(figsize=(16,6), label='30-day Rolling Avg Close', legend=True)
df.rolling(365).mean()['close'].plot(figsize=(16,6), label='1-yr Rolling Avg Close', legend=True)
```

Out[18]:
```
<AxesSubplot:title={'center':'Bitcoin Close Prices'}, xlabel='Date'>
```



Getting Data Ready for Machine Learning:

Splitting into a Train set and Test set - Now I will begin to predict future Bitcoin prices by dividing our data into two parts: train and test. Training data will contain 80 percent of our closing prices and the test set will contain the other 20 percent. The training set will be used to create the model, while the testing set will be used on our model to see if it can correctly forecast prices by using unseen values from the testing set.

In [19]:
```
import math

#Only using the prices from the Close column in our dataset
close_prices = df.filter(['close'])

#Organizing the prices into an array for easy manipulation
close_price_df = close_prices.values

#Setting the length the training dataset into 80% of the total data
#Ceiling is used to round up to the nearest whole integer
training_len = math.ceil(len(close_price_df) * 0.80)
print(training_len)
```

```
2392
```

Preprocessing Data:

Data should always be preprocessed, whether that be normalizing or scaling your data before you input it into a model. This process makes its easier for the model to digest the information.

Normalizing Data -

We'll be separating our closing prices from the dataset as we will be using these values to create the model. Normalizing our data would transform and then fit our range of closing prices to be represented as a value that lies between 0 and 1 relative to the total data using MinMaxScaler. It's important to do so as we want to change all the values in the dataset column into a common scale.

In [20]:
```
from sklearn.preprocessing import MinMaxScaler

scaler = MinMaxScaler(feature_range=(0,1))

scaled_data = scaler.fit_transform(close_price_df)
print(scaled_data)
```

```
[[0.00111246]
 [0.00076549]
 [0.00057979]
 ...
 [0.5552035 ]
 [0.53089867]
 [0.53861036]]
```

In [21]:
```
train_data = scaled_data[0: training_len, :]
print(train_data)
```

```
[[0.00111246]
 [0.00076549]
 [0.00057979]
 ...
 [0.13619706]
 [0.13279037]
 [0.13371681]]
```

In [22]:
```
x_train = [] #training features
y_train = [] #target variables

for i in range(60, len(train_data)):
    x_train.append(train_data[i-60:i, 0]) #60 day window of historical prices; appending the past 60 values: up
                                          #These are the past 60 values positioned from index [0:59]
    y_train.append(train_data[i,0]) #contains the 61th value that we want our model to predict
                                    #(which is at position 60): col 0


print('Length of list (x_train): ',len(x_train))
print('The type of this: ',type(x_train))
print(x_train[:2])
```

```
Length of list (x_train):  2332
The type of this:  <class 'list'>
[array([0.00111246, 0.00076549, 0.00057979, 0.00046219, 0.00069471,
       0.00074847, 0.00069156, 0.00067895, 0.00071152, 0.00069739,
       0.0007688 , 0.00073795, 0.00073412, 0.0007811 , 0.00067895,
       0.00072183, 0.00079339, 0.00086047, 0.00086809, 0.00084431,
       0.00084447, 0.00085834, 0.00087425, 0.00091856, 0.00102103,
       0.0010018 , 0.00102544, 0.00096656, 0.00095482, 0.00100684,
       0.00095165, 0.00095482, 0.00095955, 0.00084907, 0.00084797,
       0.00083533, 0.00083895, 0.00078141, 0.00067895, 0.0006285 ,
       0.00049766, 0.00059776, 0.00063796, 0.00062614, 0.00056072,
       0.00049733, 0.00049747, 0.00048993, 0.00052446, 0.00061431,
       0.00062771, 0.00065767, 0.00064742, 0.0006285 , 0.00061747,
       0.00054081, 0.00055993, 0.00056072, 0.00052031, 0.00041331]), array([0.00076549, 0.00057979, 0.00046219,
0.00069471, 0.00074847,
       0.00069156, 0.00067895, 0.00071152, 0.00069739, 0.0007688 ,
       0.00073795, 0.00073412, 0.0007811 , 0.00067895, 0.00072183,
       0.00079339, 0.00086047, 0.00086809, 0.00084431, 0.00084447,
       0.00085834, 0.00087425, 0.00091856, 0.00102103, 0.0010018 ,
       0.00102544, 0.00096656, 0.00095482, 0.00100684, 0.00095165,
       0.00095482, 0.00095955, 0.00084907, 0.00084797, 0.00083533,
       0.00083895, 0.00078141, 0.00067895, 0.0006285 , 0.00049766,
       0.00059776, 0.00063796, 0.00062614, 0.00056072, 0.00049733,
       0.00049747, 0.00048993, 0.00052446, 0.00061431, 0.00062771,
       0.00065767, 0.00064742, 0.0006285 , 0.00061747, 0.00054081,
       0.00055993, 0.00056072, 0.00052031, 0.00041331, 0.00041874])]
```

In [23]:
```
#preview of test data: this is a list of values that will be used as our label data
#these are all the 61st data point
print('Length of list (y_train): ', len(y_train))
print('The type of this: ',type(y_train))
print(y_train[:3])
```

```
Length of list (y_train):  2332
The type of this:  <class 'list'>
[0.00041874347384086895, 0.00044428133826022293, 0.00030092771335529434]
```

In [24]:
```
#Converting the x-training data (features) and y-training data (labels) into a numpy array.
#Numpy arrays are the format that TensorFlow accepts for neural model training in the upcoming step.
x_train, y_train = np.array(x_train), np.array(y_train)
x_train[:3]
print('x_train data (features): ', x_train.shape) #(num of rows, num of cols) <- two-dimensions
print('Now the type for x_train is: ',type(x_train))
print('y_train data (labels): ', y_train.shape)
print('Now the type for y_train is: ',type(y_train))
```

```
x_train data (features):  (2332, 60)
Now the type for x_train is:  <class 'numpy.ndarray'>
y_train data (labels):  (2332,)
Now the type for y_train is:  <class 'numpy.ndarray'>
```

Now we are reshaping the training data into three-dimensions [number of samples, time steps, features] as this is the only format accepted for LSTM modeling. (Used after neural modeling step)

In [25]:
```
'''np.reshape(the data(input), [num of samples (num of rows we have is 2332), num of time stamps(window
    size is 60), num of features is 1 (the closing price)]'''
```

```
x_train = np.reshape(x_train, (x_train.shape[0], x_train.shape[1],1))
print('x_train now has a three-dim shape: ',x_train.shape)
```

```
x_train now has a three-dim shape:  (2332, 60, 1)
```

Building the LSTM Model:

LSTM units are units of a recurrent neural network (RNN). An RNN composed of LSTM units is often called an LSTM network (or just LSTM). A common LSTM unit is composed of a cell, an input gate, an output gate and a forget gate. The cell remembers values over arbitrary time intervals and the three gates regulate the flow of information into and out of the cell.

Here is the paper that I followed: https://towardsdatascience.com/illustrated-guide-to-lstms-and-gru-s-a-step-by-step-explanation-44e9eb85bf21

In [26]:
```
from keras.models import Sequential
from keras.layers import Dense, LSTM

model = Sequential()

#Return Sequences: Accepts boolean value. Indicates whether to return the last output in the output sequence,
#or the full sequence. Default: `False`.
#inputs: A 3D tensor with shape `[batch, timesteps, feature]`.
model.add(LSTM(50, return_sequences=True, input_shape=(x_train.shape[1],1)))
model.add(LSTM(50, return_sequences=False))
model.add(Dense(25))
model.add(Dense(1)) #We want one output
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 lstm (LSTM)                 (None, 60, 50)            10400

 lstm_1 (LSTM)               (None, 50)                20200

 dense (Dense)               (None, 25)                1275

 dense_1 (Dense)             (None, 1)                 26

=================================================================
Total params: 31,901
Trainable params: 31,901
Non-trainable params: 0
_________________________________________________________________
```

Compiling the Model:

Parameters:-

- Compile: Configures the model for training.
- Optimizer: Adam optimization is a stochastic gradient descent method,
- Loss: Computes the mean of squares of errors between labels and predictions.
- Fit: Trains the model for a fixed number of epochs (iterations on a dataset).
- x: Input data
- y: Target data. Like the input data `x`
- batch_size: Integer or `None`. If unspecified, `batch_size` will default to 32.
- epochs: Integer. Number of epochs to train the model. An epoch is an iteration over the entire `x` and `y` data provided.

In [27]:
```
model.compile(optimizer='adam', loss= 'mean_squared_error')
model.fit(x_train, y_train, batch_size=1, epochs=1)
```

```
2332/2332 [==============================] - 36s 14ms/step - loss: 1.8939e-04
```
Out[27]:
```
<keras.callbacks.History at 0x1925c9c79a0>
```

Setting up the Test Data:

In [28]:
```
#Here we get the last 60 days in the dataset to estimate the price of the following day
test_data = scaled_data[training_len-60:,:]

x_test = []

#The values in the last 20% of the closing price column
y_test = close_price_df[training_len:,:]

#Creating windows of 60 test values with a shift of one value a day to the right
```

```
for i in range(60,len(test_data)):
    x_test.append(test_data[i-60:i,0])

#Convert test values into array then reshape for it to be input into model
x_test = np.array(x_test)
x_test = np.reshape(x_test, (x_test.shape[0],x_test.shape[1],1))
x_test[:1]
```

Out[28]:
```
array([[[0.15968808],
        [0.16076266],
        [0.15942628],
        [0.15687368],
        [0.15767253],
        [0.15229587],
        [0.13481724],
        [0.13271157],
        [0.12690996],
        [0.12900467],
        [0.12891118],
        [0.12667693],
        [0.12966711],
        [0.130446  ],
        [0.13123051],
        [0.12913309],
        [0.12828099],
        [0.12742281],
        [0.12484782],
        [0.12890657],
        [0.12864111],
        [0.13442587],
        [0.1342798 ],
        [0.13010676],
        [0.13034005],
        [0.13009492],
        [0.13094115],
        [0.12827201],
        [0.12578375],
        [0.12667261],
        [0.12461217],
        [0.1248542 ],
        [0.12853541],
        [0.12887659],
        [0.12626734],
        [0.11738374],
        [0.11704981],
        [0.13544992],
        [0.14466048],
        [0.14949602],
        [0.14483666],
        [0.14754083],
        [0.1440418 ],
        [0.14394498],
        [0.14491479],
        [0.1459176 ],
        [0.14450887],
        [0.14730317],
        [0.14619835],
        [0.14648766],
        [0.14501657],
        [0.1377228 ],
        [0.13785998],
        [0.14167402],
        [0.13698043],
        [0.13789277],
        [0.13777612],
        [0.13619706],
        [0.13279037],
        [0.13371681]]])
```

Test the Model and Calculate Predictions:

In [29]:
```
#Using the model we created and feeding it our testing data set to compare predicted prices with actual results
predictions = model.predict(x_test)

#Reversing our normalized data (that was predicted by the model) back into regular predicted Bitcoin prices.
predictions = scaler.inverse_transform(predictions)
predictions[:5]
```

```
19/19 [==============================] - 1s 10ms/step
```

Out[29]:
```
array([[8663.809],
       [8622.984],
       [8556.02 ],
       [8470.41 ],
       [8360.935]], dtype=float32)
```

Calculating model error using RMSE:

This is where we evaluate the pricing gap between our models predictions from the actual values in the testing set. We use the Root Mean Square Error (RMSE) to calculate the standard deviation of our predictions errors as this a commonly used metric used to evaluate predictions. The higher the RSME value the less accurate our model was in predicting future prices, so our goal here is to achieve a error rate as close to 0 as possible.

In [30]:
```
rmse = np.sqrt(np.mean(((predictions - y_test)**2)))
print(rmse)
```

```
2975.0142710181676
```

we can see the spread between predictions and actual values were large, however considering we knew that Bitcoin was fairly volatile this makes sense as pricing differs significantly on a daily basis. But let's graph this to see how well our model faired against actual prices.

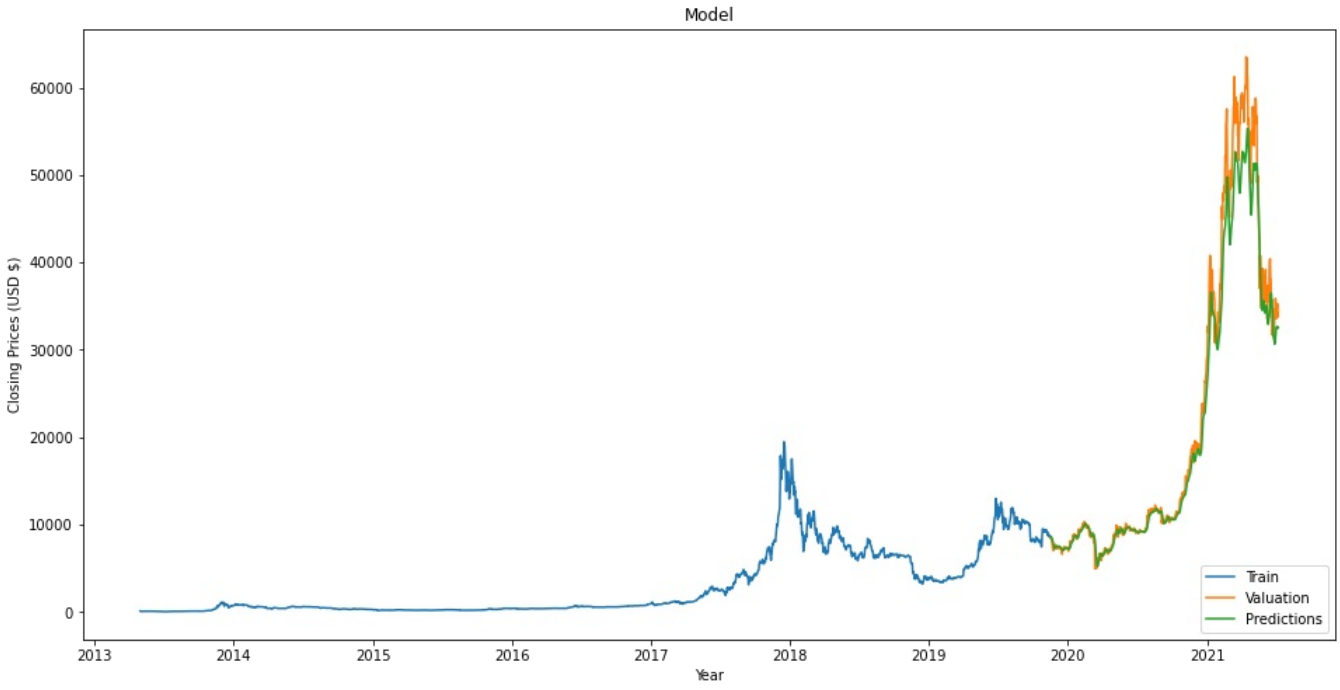
Visualizing Predictions Vs. Actuals:

In [31]:
```
data  = df.filter(['close'])

train = data[:training_len]
valuation = data[training_len:]

valuation['Predictions'] = predictions

valuation.head()
```

Out[31]:

Date	close	Predictions
2019-11-17	8577.975782	8663.808594
2019-11-18	8309.285983	8622.984375
2019-11-19	8206.145918	8556.019531
2019-11-20	8027.268243	8470.410156
2019-11-21	7642.749945	8360.934570

In [32]:
```
plt.figure(figsize=(16,8))
plt.title('Model')
plt.xlabel('Year')
plt.ylabel('Closing Prices (USD $)')
plt.plot(train)

plt.plot(valuation[['close','Predictions']])
plt.legend(['Train','Valuation','Predictions'], loc='lower right')
plt.show()
```



Final Thought

Based on what was uncovered in the above analysis for Bitcoins historical prices it appears that Bitcoin has the potential to become an

institutional investment class. Of course further examination of other cryptocurriencies should be done before determining whether or not to invest in Bitcoin or any of its competitors.

Future Work:

Next steps could include but are not limited to:

- Analyzing all high peforming crytocurrencies within one dataset.
- Investigate correlations between cyptocurrencies movements against a benchmark index (ex/S&P500).
- Use linear regression to determine how cryptocurrencies would affect a porfolios overall performance.

Github Repository for this project :

https://github.com/smritiroy14/Bitcoin-Data-Analysis