Siddharth Yadav

Everything you can do with a time series

last run 3 months ago · IPython Notebook HTML · 12,143 views
using data from multiple data sources · Public

Tags beginner tutorial time series time series analysis multiple data sources

Notebook

Aim

Since my first week on this platform, I have been fascinated by the topic of **time series analysis.** This kernel is prepared to be a container of many broad topics in the field of time series analysis. My motive is to make this the ultimate reference to time series analysis for beginners and experienced people alike.

Some important things

1. This kernel **is a work in progress so every time you see on your home feed and open it, you will surely find fresh content.**
2. I am doing this only after completing various courses in this field. I continue to study more advanced concepts to provide more knowledge and content.
3. If there is any suggestion or any specific topic you would like me to cover, kindly mention that in the comments.
4. **If you like my work, be sure to upvote**(press the like button) this kernel so it looks more relevant and meaningful to the community.

In [1]:

```
# Importing libraries
import os
import warnings
warnings.filterwarnings('ignore')
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
plt.style.use('fivethirtyeight')
# Above is a special style template for matplotlib, highly useful for v
isualizing time series data
%matplotlib inline
from pylab import rcParams
from plotly import tools
import plotly.plotly as py
from plotly.offline import init_notebook_mode, iplot
init_notebook_mode(connected=True)
import plotly.graph_objs as go
import plotly.figure_factory as ff
import statsmodels.api as sm
from numpy.random import normal, seed
from scipy.stats import norm
from statsmodels.tsa.arima_model import ARMA
from statsmodels.tsa.stattools import adfuller
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf
from statsmodels.tsa.arima_process import ArmaProcess
from statsmodels.tsa.arima_model import ARIMA
import math
from sklearn.metrics import mean_squared_error
```

```
print(os.listdir("../input"))
```

```
['historical-hourly-weather-data', 'stock-time-series-20050101-to-201
71231']
```


- 1. Introduction to date and time
  - 1.1 Importing time series data
  - 1.2 Cleaning and preparing time series data
  - 1.3 Visualizing the datasets
  - 1.4 Timestamps and Periods
  - 1.5 Using date_range
  - 1.6 Using to_datetime
  - 1.7 Shifting and lags
  - 1.8 Resampling
- 2. Finance and Statistics
  - 2.1 Percent change
  - 2.2 Stock returns
  - 2.3 Absolute change in successive rows
  - 2.4 Comaring two or more time series
  - 2.5 Window functions
  - 2.6 OHLC charts
  - 2.7 Candlestick charts
  - 2.8 Autocorrelation and Partial Autocorrelation
- 3. Time series decomposition and Random Walks
  - 3.1 Trends, Seasonality and Noise
  - 3.2 White Noise
  - 3.3 Random Walk
  - 3.4 Stationarity
- 4. Modelling using statsmodels
  - 4.1 AR models
  - 4.2 MA models
  - 4.3 ARMA models
  - 4.4 ARIMA models
  - 4.5 VAR models
  - 4.6 State space methods
    - 4.6.1 SARIMA models
    - 4.6.2 Unobserved components
    - 4.6.3 Dynamic Factor models

1. Introduction to date and time

1.1 Importing time series data

How to import data?

First, we import all the datasets needed for this kernel. The required time series column is imported as a datetime column using **parse_dates** parameter and is also selected as index of the dataframe using **index_col** parameter.

Data being used:-

1. Google Stocks Data
2. Humidity in different world cities
3. Microsoft Stocks Data
4. Pressure in different world cities

In [2]:

```
google = pd.read_csv('../input/stock-time-series-20050101-to-20171231/GOOGL_2006-01-01_to_2018-01-01.csv', index_col='Date', parse_dates=['Date'])
google.head()
```

Out[2]:

	Open	High	Low	Close	Volume	Name
Date						
2006-01-03	211.47	218.05	209.32	217.83	13137450	GOOGL
2006-01-04	222.17	224.70	220.09	222.84	15292353	GOOGL
2006-01-05	223.22	226.00	220.97	225.85	10815661	GOOGL
2006-01-06	228.66	235.49	226.85	233.06	17759521	GOOGL
2006-01-09	233.44	236.94	230.70	233.68	12795837	GOOGL

In [3]:

```
humidity = pd.read_csv('../input/historical-hourly-weather-data/humidity.csv', index_col='datetime', parse_dates=['datetime'])
humidity.tail()
```

Out[3]:

	Vancouver	Portland	San Francisco	Seattle	Los Angeles	San Diego	Las Vegas	Phoenix	Albuqu
datetime									
2017-11-29	NaN	81.0	NaN	93.0	24.0	72.0	18.0	68.0	37.0

20:00:00									
2017-11-29 21:00:00	NaN	71.0	NaN	87.0	21.0	72.0	18.0	73.0	34.0
2017-11-29 22:00:00	NaN	71.0	NaN	93.0	23.0	68.0	17.0	60.0	32.0
2017-11-29 23:00:00	NaN	71.0	NaN	87.0	14.0	63.0	17.0	33.0	30.0
2017-11-30 00:00:00	NaN	76.0	NaN	75.0	56.0	72.0	17.0	23.0	34.0

1.2 Cleaning and preparing time series data

How to prepare data?

Google stocks data doesn't have any missing values but humidity data does have its fair share of missing values. It is cleaned using **fillna()** method with **ffill** parameter which propagates last valid observation to fill gaps

In [4]:

```
humidity = humidity.iloc[1:]
humidity = humidity.fillna(method='ffill')
humidity.head()
```

Out[4]:

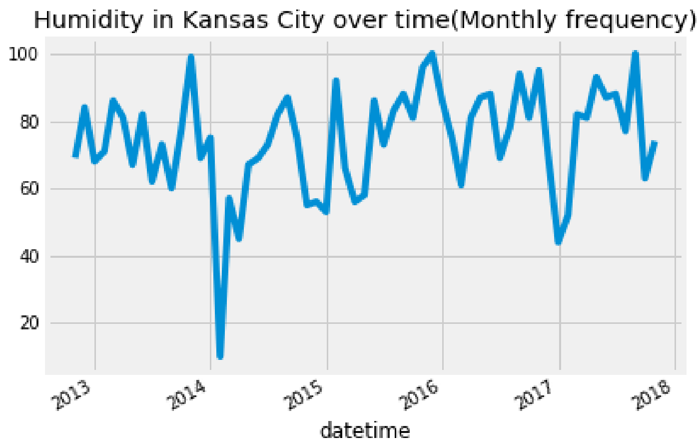
	Vancouver	Portland	San Francisco	Seattle	Los Angeles	San Diego	Las Vegas	Phoenix	Albuqu
datetime									
2012-10-01 13:00:00	76.0	81.0	88.0	81.0	88.0	82.0	22.0	23.0	50.0
2012-10-01 14:00:00	76.0	80.0	87.0	80.0	88.0	81.0	21.0	23.0	49.0
2012-10-01 15:00:00	76.0	80.0	86.0	80.0	88.0	81.0	21.0	23.0	49.0
2012-10-01 16:00:00	77.0	80.0	85.0	79.0	88.0	81.0	21.0	23.0	49.0

2012-10-01 17:00:00	78.0	79.0	84.0	79.0	88.0	80.0	21.0	24.0	49.0

1.3 Visualizing the datasets

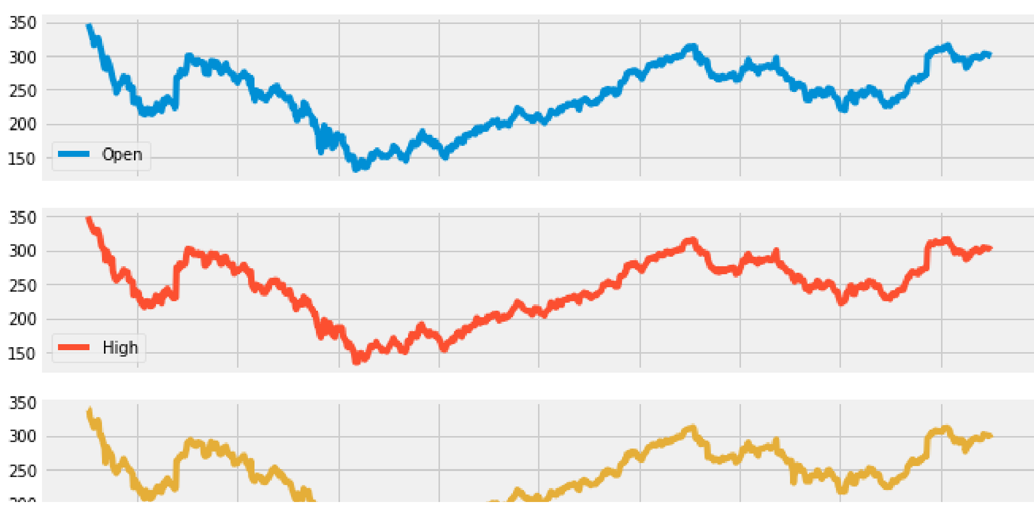
In [5]:

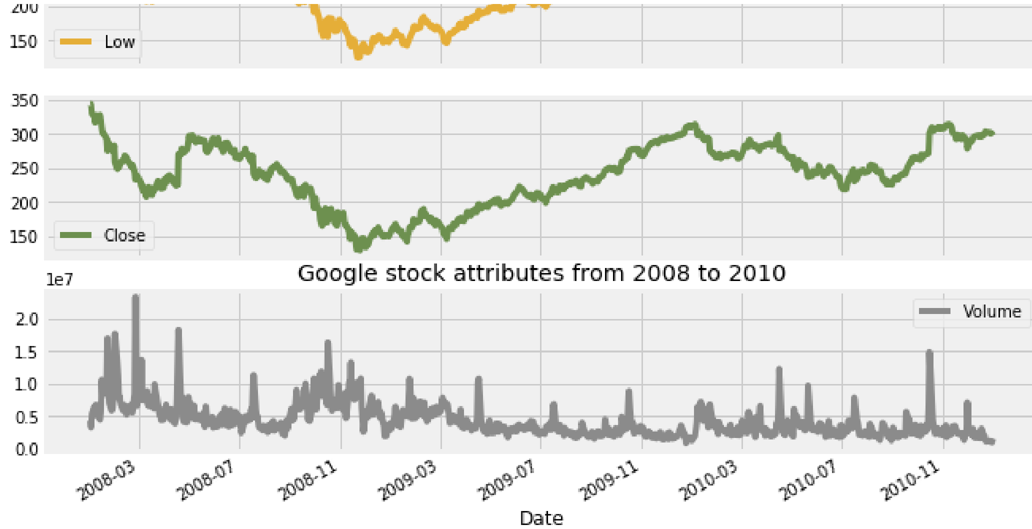
```
humidity["Kansas City"].asfreq('M').plot() # asfreq method is used to
 convert a time series to a specified frequency. Here it is monthly fre
quency.
plt.title('Humidity in Kansas City over time(Monthly frequency)')
plt.show()
```



In [6]:

```
google['2008':'2010'].plot(subplots=True, figsize=(10,12))
plt.title('Google stock attributes from 2008 to 2010')
plt.savefig('stocks.png')
plt.show()
```





1.4 Timestamps and Periods

What are timestamps and periods and how are they useful?

Timestamps are used to represent a point in time. Periods represent an interval in time. Periods can used to check if a specific event in the given period. They can also be converted to each other's form.

In [7]:
```
# Creating a Timestamp
timestamp = pd.Timestamp(2017, 1, 1, 12)
timestamp
```

Out[7]:
```
Timestamp('2017-01-01 12:00:00')
```

In [8]:
```
# Creating a period
period = pd.Period('2017-01-01')
period
```

Out[8]:
```
Period('2017-01-01', 'D')
```

In [9]:
```
# Checking if the given timestamp exists in the given period
period.start_time < timestamp < period.end_time
```

Out[9]:
```
True
```

In [10]:
```
# Converting timestamp to period
```

```
new_period = timestamp.to_period(freq='H')
new_period
```

Out[10]:

```
Period('2017-01-01 12:00', 'H')
```

In [11]:

```
# Converting period to timestamp
new_timestamp = period.to_timestamp(freq='H', how='start')
new_timestamp
```

Out[11]:

```
Timestamp('2017-01-01 00:00:00')
```

1.5 Using date_range

What is date_range and how is it useful?

date_range is a method that returns a fixed frequency datetimeindex. It is quite useful when creating your own time series attribute for pre-existing data or arranging the whole data around the time series attribute created by you.

In [12]:

```
# Creating a datetimeindex with daily frequency
dr1 = pd.date_range(start='1/1/18', end='1/9/18')
dr1
```

Out[12]:

```
DatetimeIndex(['2018-01-01', '2018-01-02', '2018-01-03', '2018-01-0
4',
               '2018-01-05', '2018-01-06', '2018-01-07', '2018-01-0
8',
               '2018-01-09'],
              dtype='datetime64[ns]', freq='D')
```

In [13]:

```
# Creating a datetimeindex with monthly frequency
dr2 = pd.date_range(start='1/1/18', end='1/1/19', freq='M')
dr2
```

Out[13]:

```
DatetimeIndex(['2018-01-31', '2018-02-28', '2018-03-31', '2018-04-3
0',
               '2018-05-31', '2018-06-30', '2018-07-31', '2018-08-3
1',
               '2018-09-30', '2018-10-31', '2018-11-30', '2018-12-3
1'],
              dtype='datetime64[ns]', freq='M')
```

In [14]:

```
# Creating a datetimeindex without specifying start date and using peri
ods
dr3 = pd.date_range(end='1/4/2014', periods=8)
dr3
```

Out[14]:

```
DatetimeIndex(['2013-12-28', '2013-12-29', '2013-12-30', '2013-12-3
1',
               '2014-01-01', '2014-01-02', '2014-01-03', '2014-01-0
4'],
              dtype='datetime64[ns]', freq='D')
```

In [15]:

```
# Creating a datetimeindex specifying start date , end date and periods
dr4 = pd.date_range(start='2013-04-24', end='2014-11-27', periods=3)
dr4
```

Out[15]:

```
DatetimeIndex(['2013-04-24', '2014-02-09', '2014-11-27'], dtype='date
time64[ns]', freq=None)
```

1.6 Using to_datetime

pandas.to_datetime() is used for converting arguments to datetime. Here, a DataFrame is converted to a datetime series.

In [16]:

```
df = pd.DataFrame({'year': [2015, 2016], 'month': [2, 3], 'day': [4, 5
]})
df
```

Out[16]:

	year	month	day
0	2015	2	4
1	2016	3	5

In [17]:

```
df = pd.to_datetime(df)
df
```

Out[17]:

```
0   2015-02-04
```

```
1    2016-03-05
dtype: datetime64[ns]
```

In [18]:

```
df = pd.to_datetime('01-01-2017')
df
```

Out[18]:

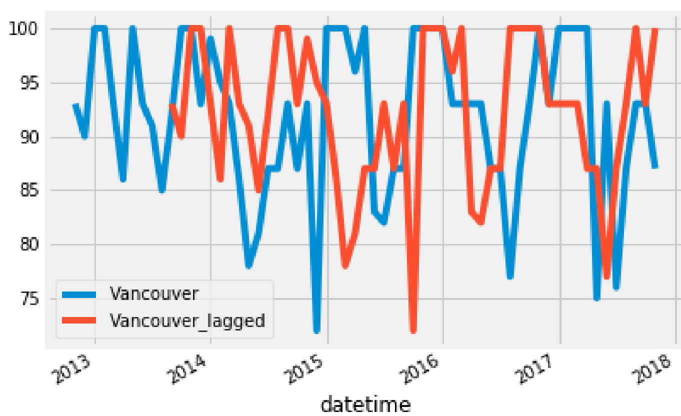
```
Timestamp('2017-01-01 00:00:00')
```

1.7 Shifting and lags

We can shift index by desired number of periods with an optional time frequency. This is useful when comparing the time series with a past of itself

In [19]:

```
humidity["Vancouver"].asfreq('M').plot(legend=True)
shifted = humidity["Vancouver"].asfreq('M').shift(10).plot(legend=True
)
shifted.legend(['Vancouver','Vancouver_lagged'])
plt.show()
```



1.8 Resampling

Upsampling - Time series is resampled from low frequency to high frequency(Monthly to daily frequency). It involves filling or interpolating missing data

Downsampling - Time series is resampled from high frequency to low frequency(Weekly to monthly frequency). It involves aggregation of existing data.

In [20]:

```
# Let's use pressure data to demonstrate this
pressure = pd.read_csv('../input/historical-hourly-weather-data/pressu
```

```
re.csv', index_col='datetime', parse_dates=['datetime'])
pressure.tail()
```

Out[20]:

	Vancouver	Portland	San Francisco	Seattle	Los Angeles	San Diego	Las Vegas	Phoenix	Albu
datetime									
2017-11-29 20:00:00	NaN	1031.0	NaN	1030.0	1016.0	1017.0	1021.0	1018.0	1025
2017-11-29 21:00:00	NaN	1030.0	NaN	1030.0	1016.0	1017.0	1020.0	1018.0	1024
2017-11-29 22:00:00	NaN	1030.0	NaN	1029.0	1015.0	1016.0	1020.0	1017.0	1024
2017-11-29 23:00:00	NaN	1029.0	NaN	1028.0	1016.0	1016.0	1020.0	1016.0	1024
2017-11-30 00:00:00	NaN	1029.0	NaN	1028.0	1015.0	1017.0	1019.0	1016.0	1024

Sigh! A lot of cleaning is required.

In [21]:

```
pressure = pressure.iloc[1:]
pressure = pressure.fillna(method='ffill')
pressure.tail()
```

Out[21]:

	Vancouver	Portland	San Francisco	Seattle	Los Angeles	San Diego	Las Vegas	Phoenix	Albu
datetime									
2017-11-29 20:00:00	1021.0	1031.0	1013.0	1030.0	1016.0	1017.0	1021.0	1018.0	1025
2017-11-29 21:00:00	1021.0	1030.0	1013.0	1030.0	1016.0	1017.0	1020.0	1018.0	1024
2017-11-29 22:00:00	1021.0	1030.0	1013.0	1029.0	1015.0	1016.0	1020.0	1017.0	1024

2017-11-29 23:00:00	1021.0	1029.0	1013.0	1028.0	1016.0	1016.0	1020.0	1016.0	1024
2017-11-30 00:00:00	1021.0	1029.0	1013.0	1028.0	1015.0	1017.0	1019.0	1016.0	1024

In [22]:

```
pressure = pressure.fillna(method='bfill')
pressure.head()
```

Out[22]:

	Vancouver	Portland	San Francisco	Seattle	Los Angeles	San Diego	Las Vegas	Phoenix	Albu
datetime									
2012-10-01 13:00:00	807.0	1024.0	1009.0	1027.0	1013.0	1013.0	1018.0	1013.0	1024
2012-10-01 14:00:00	807.0	1024.0	1009.0	1027.0	1013.0	1013.0	1018.0	1013.0	1024
2012-10-01 15:00:00	807.0	1024.0	1009.0	1028.0	1013.0	1013.0	1018.0	1013.0	1024
2012-10-01 16:00:00	807.0	1024.0	1009.0	1028.0	1013.0	1013.0	1018.0	1013.0	1024
2012-10-01 17:00:00	807.0	1024.0	1009.0	1029.0	1013.0	1013.0	1018.0	1013.0	1024

First, we used **ffill** parameter which propagates last valid observation to fill gaps. Then we use **bfill** to propogate next valid observation to fill gaps.

In [23]:

```
# Shape before resampling(downsampling)
pressure.shape
```

Out[23]:

```
(45252, 36)
```

In [24]:

```
# We downsample from hourly to 3 day frequency aggregated using mean
pressure = pressure.resample('3D').mean()
pressure.head()
```

Out[24]:

	Vancouver	Portland	San Francisco	Seattle	Los Angeles	San Diego
datetime						
2012-10-01 13:00:00	946.652778	1022.597222	1010.666667	1030.666667	1011.472222	1011.875000
2012-10-04 13:00:00	1018.875000	1022.819444	1016.027778	1027.527778	1016.208333	1016.888889
2012-10-07 13:00:00	1014.125000	1016.652778	1016.527778	1017.472222	1013.388889	1014.347222
2012-10-10 13:00:00	1011.375000	1014.513889	1014.416667	1017.472222	1009.916667	1013.750000
2012-10-13 13:00:00	1010.208333	1018.694444	1021.888889	1016.152778	1017.972222	1018.347222

In [25]:

```
# Shape after resampling(downsampling)
pressure.shape
```

Out[25]:

```
(629, 36)
```

Much less rows are left. Now, we will upsample from 3 day frequency to daily frequency

In [26]:

```
pressure = pressure.resample('D').pad()
pressure.head()
```

Out[26]:

	Vancouver	Portland	San Francisco	Seattle	Los Angeles	San Diego
datetime						
2012-10-01	NaN	NaN	NaN	NaN	NaN	NaN
2012-	946.652778	1022.597222	1010.666667	1030.666667	1011.472222	1011.875000

10-02						
2012-10-03	946.652778	1022.597222	1010.666667	1030.666667	1011.472222	1011.875000
2012-10-04	946.652778	1022.597222	1010.666667	1030.666667	1011.472222	1011.875000
2012-10-05	1018.875000	1022.819444	1016.027778	1027.527778	1016.208333	1016.888889

In [27]:

```
# Shape after resampling(upsampling)
pressure.shape
```

Out[27]:

```
(1885, 36)
```

Again an increase in number of rows. Resampling is cool when used properly.

2. Finance and statistics

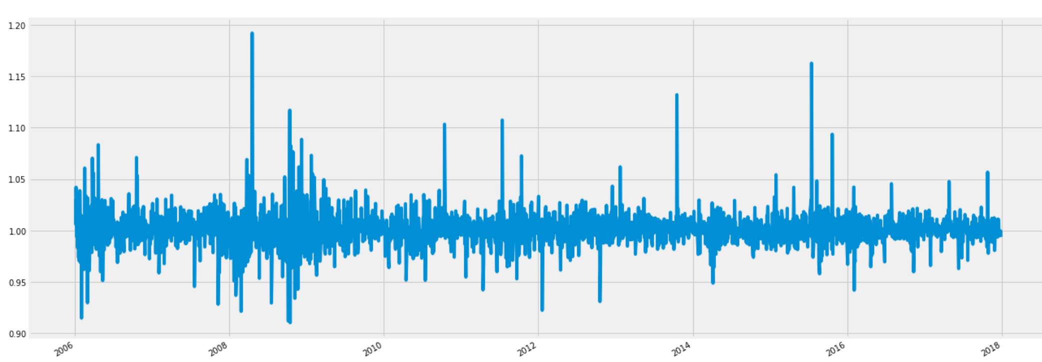
2.1 Percent change

In [28]:

```
google['Change'] = google.High.div(google.High.shift())
google['Change'].plot(figsize=(20,8))
```

Out[28]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f3db5d4ca58>
```



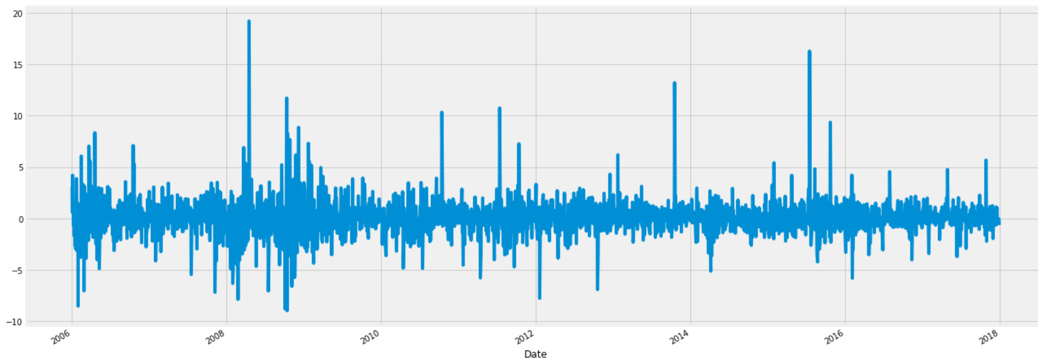
2.2 Stock returns

In [29]:

```
google['Return'] = google.Change.sub(1).mul(100)
google['Return'].plot(figsize=(20,8))
```

Out[29]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f3db5d2c8d0>
```

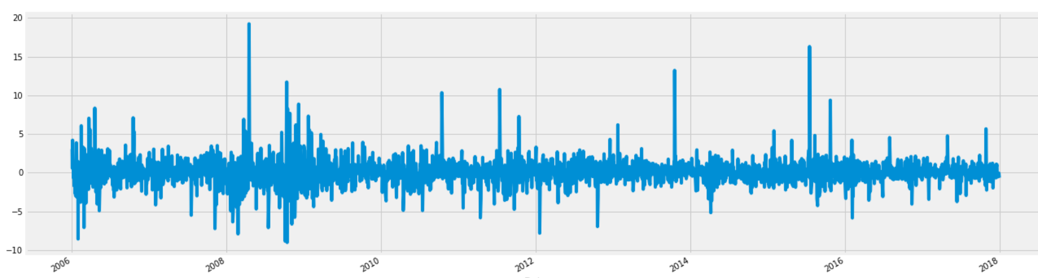


In [30]:

```
google.High.pct_change().mul(100).plot(figsize=(20,6)) # Another way t
o calculate returns
```

Out[30]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f3db4c949b0>
```



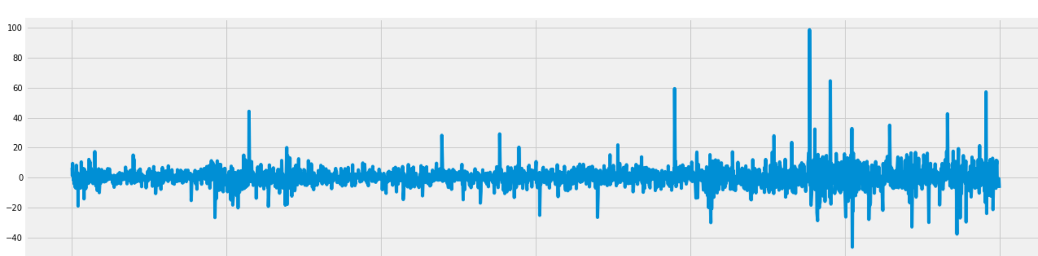
2.3 Absolute change in successive rows

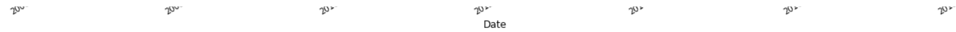
In [31]:

```
google.High.diff().plot(figsize=(20,6))
```

Out[31]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f3db4b7b6a0>
```



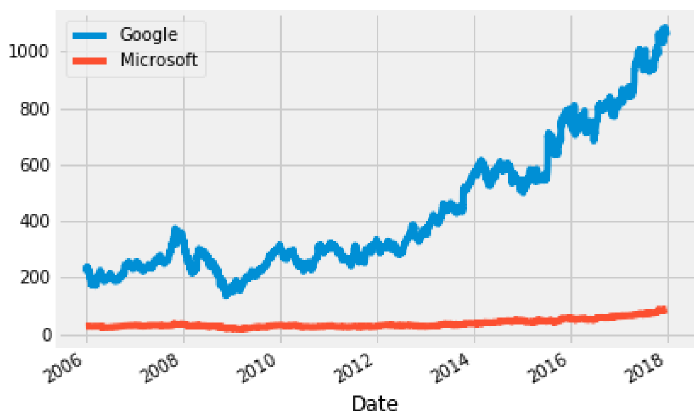


2.4 Comaring two or more time series

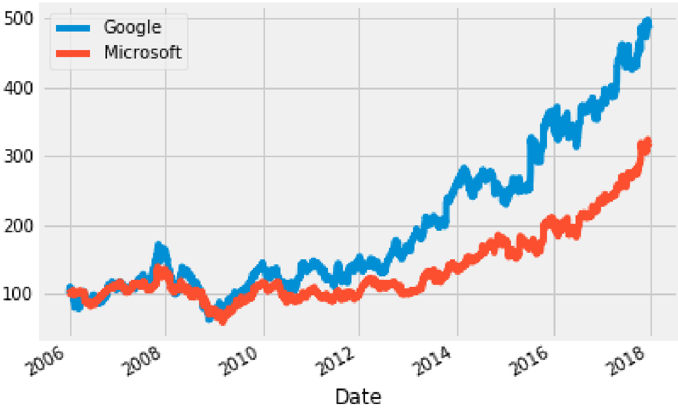
We will compare 2 time series by normalizing them. This is achieved by dividing each time series element of all time series by the first element. This way both series start at the same point and can be easily compared.

In [32]:
```
# We choose microsoft stocks to compare them with google
microsoft = pd.read_csv('../input/stock-time-series-20050101-to-201712
31/MSFT_2006-01-01_to_2018-01-01.csv', index_col='Date', parse_dates=[
'Date'])
```

In [33]:
```
# Plotting before normalization
google.High.plot()
microsoft.High.plot()
plt.legend(['Google','Microsoft'])
plt.show()
```



In [34]:
```
# Normalizing and comparison
# Both stocks start from 100
normalized_google = google.High.div(google.High.iloc[0]).mul(100)
normalized_microsoft = microsoft.High.div(microsoft.High.iloc[0]).mul(
100)
normalized_google.plot()
normalized_microsoft.plot()
plt.legend(['Google','Microsoft'])
plt.show()
```



You can clearly see how google outperforms microsoft over time.

2.5 Window functions

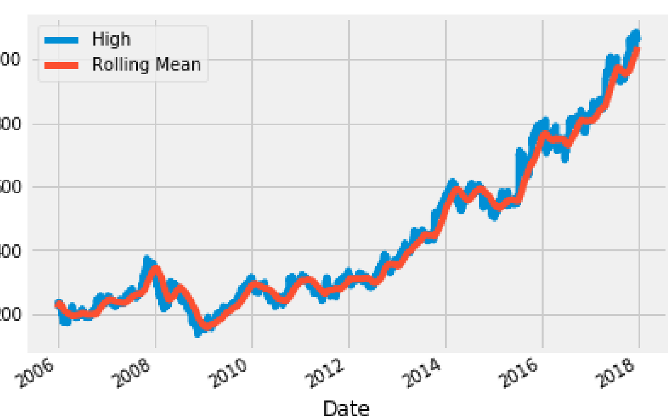
Window functions are used to identify sub periods, calculates sub-metrics of sub-periods.

Rolling - Same size and sliding

Expanding - Contains all prior values

In [35]:

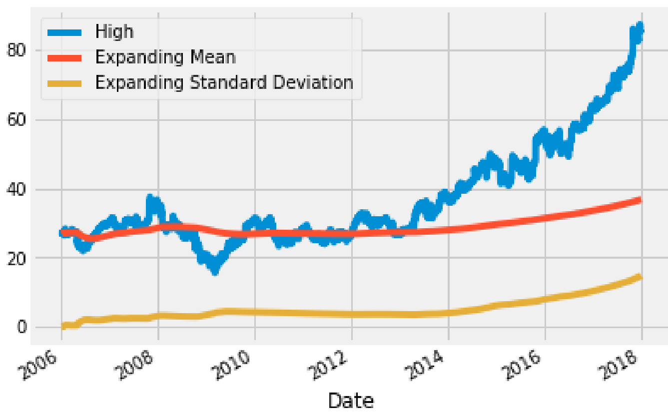
```
# Rolling window functions
rolling_google = google.High.rolling('90D').mean()
google.High.plot()
rolling_google.plot()
plt.legend(['High','Rolling Mean'])
# Plotting a rolling mean of 90 day window with original High attribute
 of google stocks
plt.show()
```



Now, observe that rolling mean plot is a smoother version of the original plot.

In [36]:

```
# Expanding window functions
microsoft_mean = microsoft.High.expanding().mean()
microsoft_std = microsoft.High.expanding().std()
microsoft.High.plot()
microsoft_mean.plot()
microsoft_std.plot()
plt.legend(['High','Expanding Mean','Expanding Standard Deviation'])
plt.show()
```

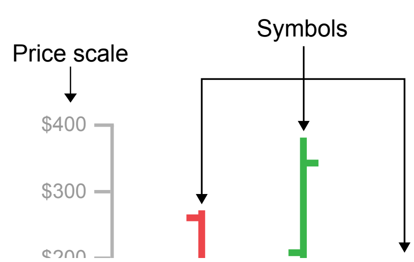


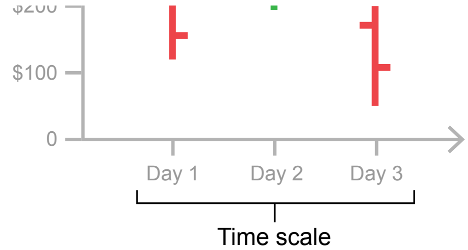
2.6 OHLC charts

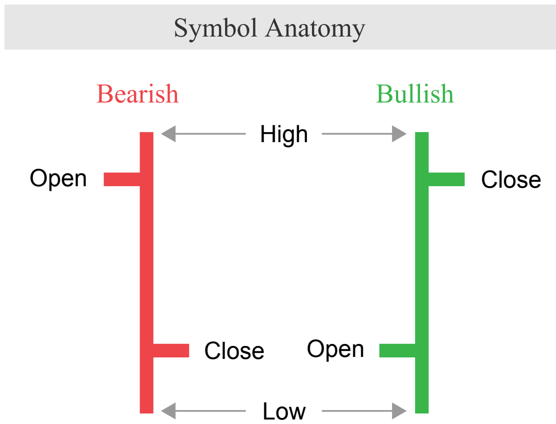
An OHLC chart is any type of price chart that shows the open, high, low and close price of a certain time period. Open-high-low-close Charts (or OHLC Charts) are used as a trading tool to visualise and analyse the price changes over time for securities, currencies, stocks, bonds, commodities, etc. OHLC Charts are useful for interpreting the day-to-day sentiment of the market and forecasting any future price changes through the patterns produced.

The y-axis on an OHLC Chart is used for the price scale, while the x-axis is the timescale. On each single time period, an OHLC Charts plots a symbol that represents two ranges: the highest and lowest prices traded, and also the opening and closing price on that single time period (for example in a day). On the range symbol, the high and low price ranges are represented by the length of the main vertical line. The open and close prices are represented by the vertical positioning of tick-marks that appear on the left (representing the open price) and on right (representing the close price) sides of the high-low vertical line.

Colour can be assigned to each OHLC Chart symbol, to distinguish whether the market is "bullish" (the closing price is higher then it opened) or "bearish" (the closing price is lower then it opened).







Source: Datavizcatalogue (https://datavizcatalogue.com/methods/OHLC_chart.html)

In [37]:

```
# OHLC chart of June 2008
trace = go.Ohlc(x=google['06-2008'].index,
                open=google['06-2008'].Open,
                high=google['06-2008'].High,
                low=google['06-2008'].Low,
                close=google['06-2008'].Close)
data = [trace]
iplot(data, filename='simple_ohlc')
```

In [38]:

```
# OHLC chart of 2008
trace = go.Ohlc(x=google['2008'].index,
                open=google['2008'].Open,
                high=google['2008'].High,
                low=google['2008'].Low,
                close=google['2008'].Close)
data = [trace]
iplot(data, filename='simple_ohlc')
```

In [39]:

```
# OHLC chart of 2008
trace = go.Ohlc(x=google.index,
                open=google.Open,
                high=google.High,
                low=google.Low,
                close=google.Close)
```

```
data = [trace]
iplot(data, filename='simple_ohlc')
```

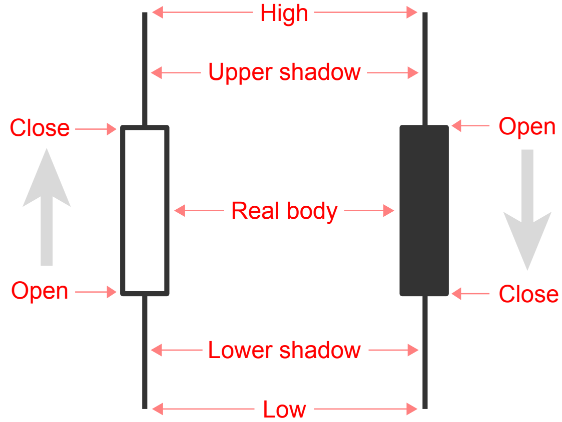
2.7 Candlestick charts

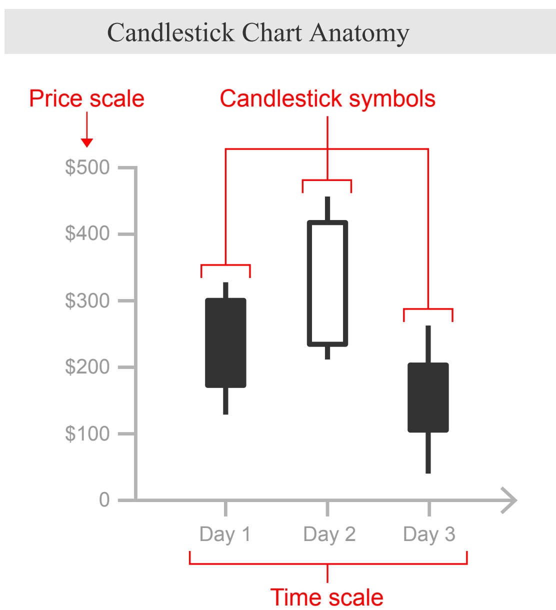
This type of chart is used as a trading tool to visualise and analyse the price movements over time for securities, derivatives, currencies, stocks, bonds, commodities, etc. Although the symbols used in Candlestick Charts resemble a Box Plot, they function differently and therefore, are not to be confused with one another.

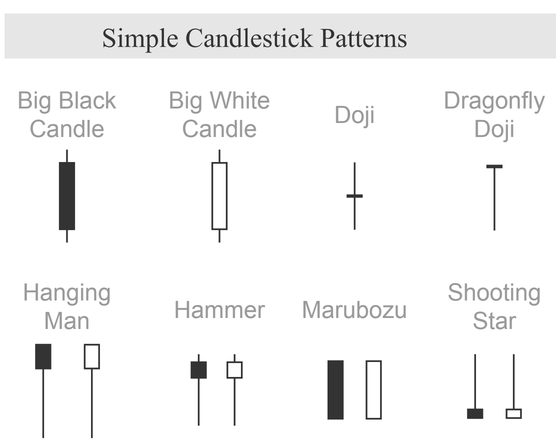
Candlestick Charts display multiple bits of price information such as the open price, close price, highest price and lowest price through the use of candlestick-like symbols. Each symbol represents the compressed trading activity for a single time period (a minute, hour, day, month, etc). Each Candlestick symbol is plotted along a time scale on the x-axis, to show the trading activity over time.

The main rectangle in the symbol is known as the real body, which is used to display the range between the open and close price of that time period. While the lines extending from the bottom and top of the real body is known as the lower and upper shadows (or wick). Each shadow represents the highest or lowest price traded during the time period represented. When the market is Bullish (the closing price is higher than it opened), then the body is coloured typically white or green. But when the market is Bearish (the closing price is lower than it opened), then the body is usually coloured either black or red.

Bullish Candlestick   Bearish Candlestick







Candlestick Charts are great for detecting and predicting market trends over time and are useful for interpreting the day-to-day sentiment of the market, through each candlestick symbol's colouring and shape. For example, the longer the body is, the more intense the selling or buying pressure is. While, a very short body, would indicate that there is very little price movement in that time period and represents consolidation.

Candlestick Charts help reveal the market psychology (the fear and greed experienced by sellers and buyers) through the various indicators, such as shape and colour, but also by the many identifiable patterns that can be found in Candlestick Charts. In total, there are 42 recognised patterns that are divided into simple and complex patterns. These patterns found in Candlestick Charts are useful for displaying price relationships and can be used for predicting the possible future movement of the market. You can find a list and description of each pattern here.

Please bear in mind, that Candlestick Charts don't express the events taking place between the open and close price - only the relationship between the two prices. So you can't tell how volatile trading was within that single time period.

Source: Datavizcatalogue (https://datavizcatalogue.com/methods/candlestick_chart.html)

In [40]:

```
# Candlestick chart of march 2008
trace = go.Candlestick(x=google['03-2008'].index,
                open=google['03-2008'].Open,
                high=google['03-2008'].High,
                low=google['03-2008'].Low,
                close=google['03-2008'].Close)
data = [trace]
iplot(data, filename='simple_candlestick')
```

In [41]:

```
# Candlestick chart of 2008
trace = go.Candlestick(x=google['2008'].index,
                open=google['2008'].Open,
```

```
                high=google['2008'].High,
                low=google['2008'].Low,
                close=google['2008'].Close)
data = [trace]
iplot(data, filename='simple_candlestick')
```

In [42]:

```
# Candlestick chart of 2006-2018
trace = go.Candlestick(x=google.index,
                open=google.Open,
                high=google.High,
                low=google.Low,
                close=google.Close)
data = [trace]
iplot(data, filename='simple_candlestick')
```

2.8 Autocorrelation and Partial Autocorrelation

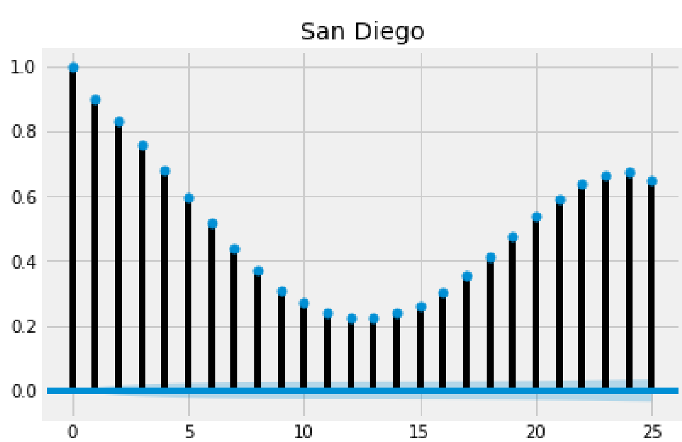
- Autocorrelation - The autocorrelation function (ACF) measures how a series is correlated with itself at different lags.
- Partial Autocorrelation - The partial autocorrelation function can be interpreted as a regression of the series against its past lags. The terms can be interpreted the same way as a standard linear regression, that is the contribution of a change in that particular lag while holding others constant.

Source: Quora (https://www.quora.com/What-is-the-difference-among-auto-correlation-partial-auto-correlation-and-inverse-auto-correlation-while-modelling-an-ARIMA-series)

Autocorrelation

In [43]:

```
# Autocorrelation of humidity of San Diego
plot_acf(humidity["San Diego"],lags=25,title="San Diego")
plt.show()
```

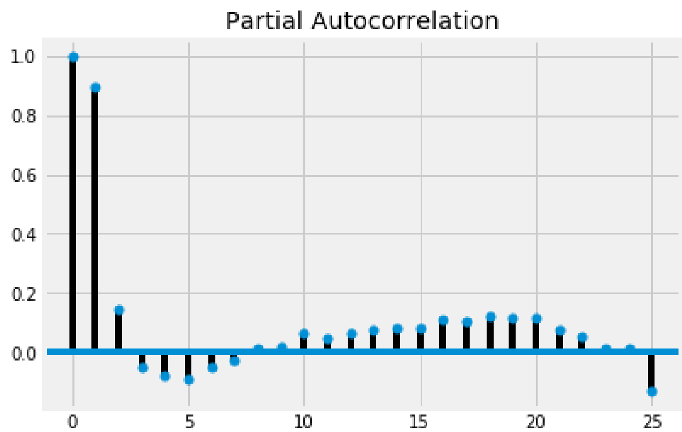


As all lags are either close to 1 or at least greater than the confidence interval, they are statistically significant.

Partial Autocorrelation

In [44]:

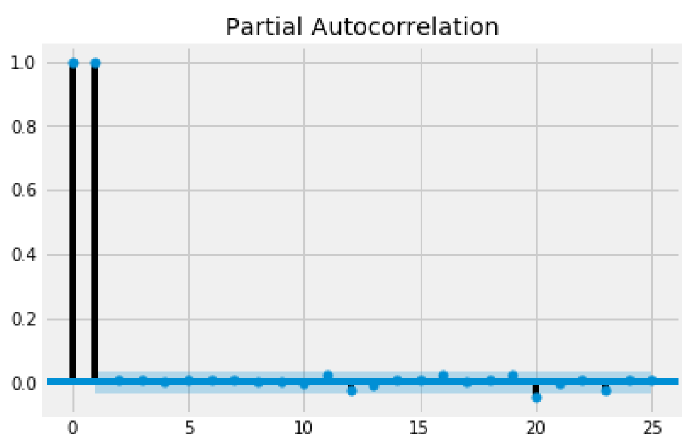
```
# Partial Autocorrelation of humidity of San Diego
plot_pacf(humidity["San Diego"],lags=25)
plt.show()
```



Though it is statistically signficant, partial autocorrelation after first 2 lags is very low.

In [45]:

```
# Partial Autocorrelation of closing price of microsoft stocks
plot_pacf(microsoft["Close"],lags=25)
plt.show()
```



Here, only 0th, 1st and 20th lag are statistically significant.

3. Time series decomposition and Random walks

3.1. Trends, seasonality and noise

These are the components of a time series

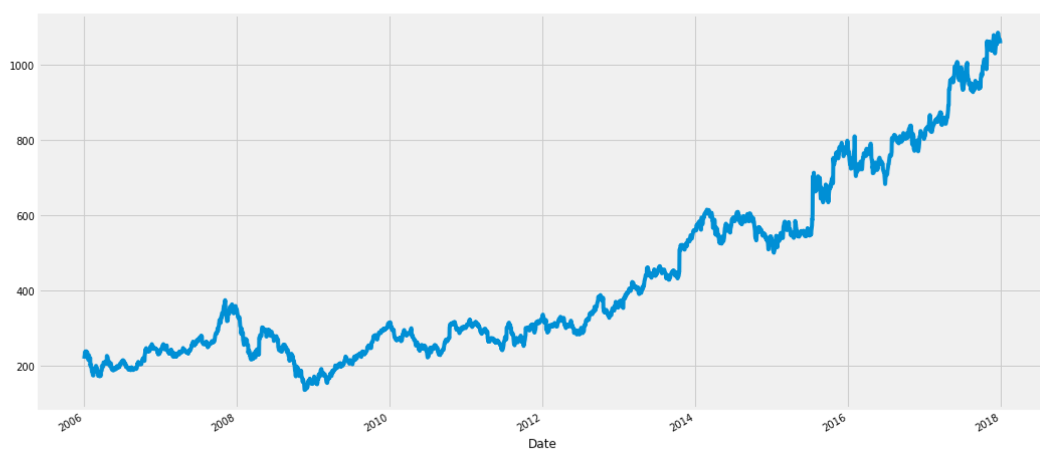
- Trend - Consistent upwards or downwards slope of a time series
- Seasonality - Clear periodic pattern of a time series(like sine funtion)
- Noise - Outliers or missing values

In [46]:

```
# Let's take Google stocks High for this
google["High"].plot(figsize=(16,8))
```

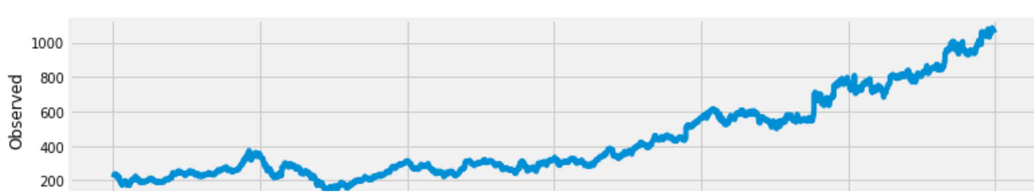
Out[46]:

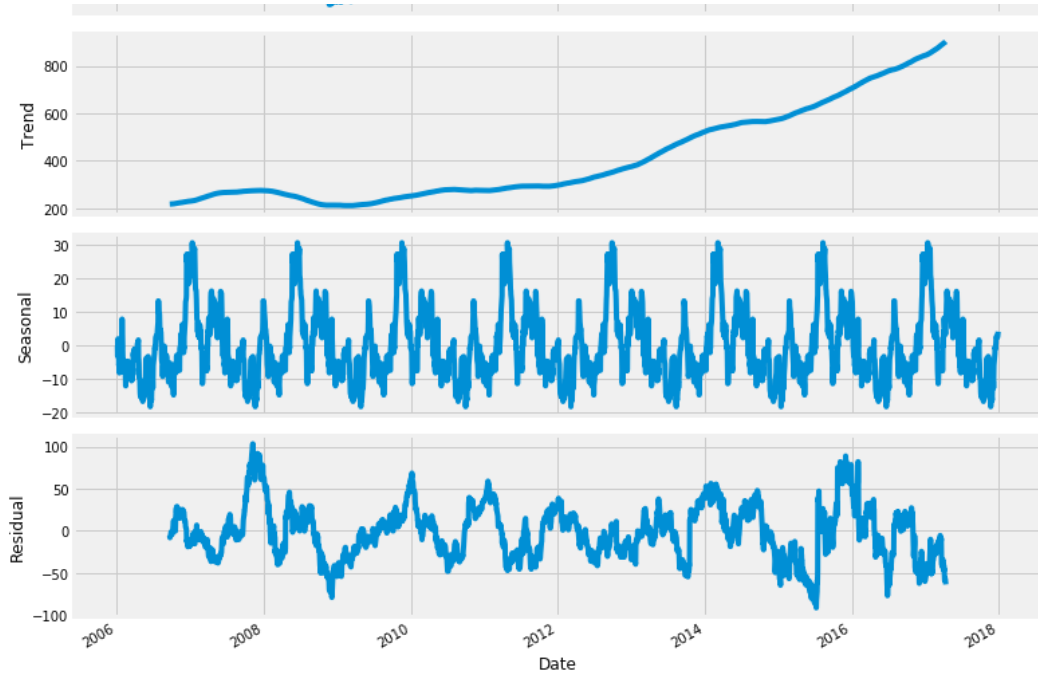
```
<matplotlib.axes._subplots.AxesSubplot at 0x7f3db6f0e978>
```



In [47]:

```
# Now, for decomposition...
rcParams['figure.figsize'] = 11, 9
decomposed_google_volume = sm.tsa.seasonal_decompose(google["High"],fr
eq=360) # The frequncy is annual
figure = decomposed_google_volume.plot()
plt.show()
```





- There is clearly an upward trend in the above plot.
- You can also see the uniform seasonal change.
- Non-uniform noise that represent outliers and missing values

3.2. White noise

White noise has...

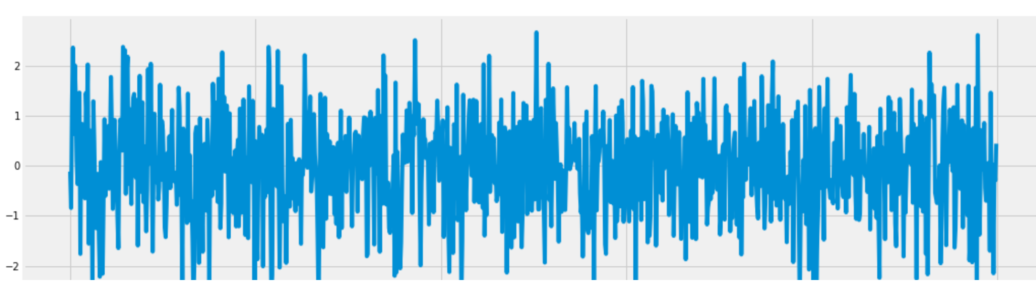
- Constant mean
- Constant variance
- Zero auto-correlation at all lags

In [48]:

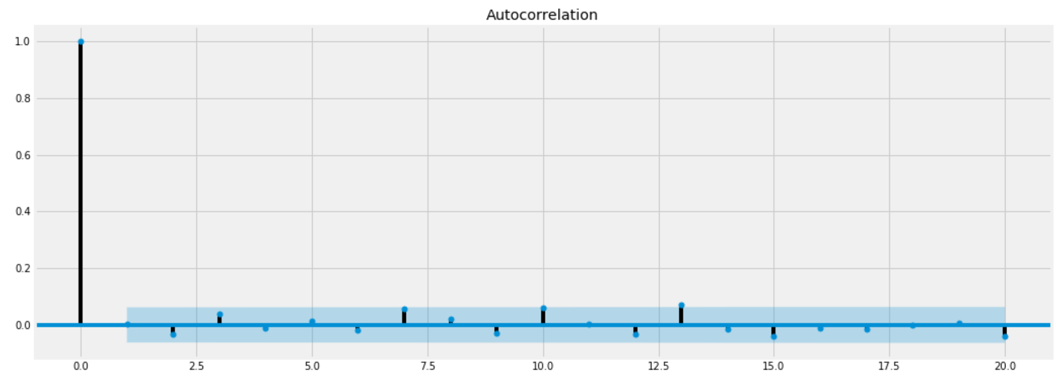
```
# Plotting white noise
rcParams['figure.figsize'] = 16, 6
white_noise = np.random.normal(loc=0, scale=1, size=1000)
# loc is mean, scale is variance
plt.plot(white_noise)
```

Out[48]:

```
[<matplotlib.lines.Line2D at 0x7f3db4413be0>]
```



In [49]:
```
# Plotting autocorrelation of white noise
plot_acf(white_noise,lags=20)
plt.show()
```



See how all lags are statistically insigficant as they lie inside the confidence interval(shaded portion).

3.3. Random Walk

A random walk is a mathematical object, known as a stochastic or random process, that describes a path that consists of a succession of random steps on some mathematical space such as the integers.

In general if we talk about stocks, Today's Price = Yesterday's Price + Noise

$P_t = P_{t-1} + \varepsilon_t$

Random walks can't be forecasted because well, noise is random.

Random Walk with Drift(drift(μ) is zero-mean)

$\mathbf{P_t - P_{t-1} = \mu + \varepsilon_t}$

Regression test for random walk

$\mathbf{P_t = \alpha + \beta P_{t-1} + \varepsilon_t}$

Equivalent to $\mathbf{P_t - P_{t-1} = \alpha + \beta P_{t-1} + \varepsilon_t}$

Test:

$\mathbf{H_0}$**: β = 1 (This is a random walk)**

H_1: β < 1 (This is not a random walk)

Dickey-Fuller Test:

H_0: β = 0 (This is a random walk)

H_1: β < 0 (This is not a random walk)

Augmented Dickey-Fuller test

An augmented Dickey–Fuller test (ADF) tests the null hypothesis that a unit root is present in a time series sample. It is basically Dickey-Fuller test with more lagged changes on RHS.

In [50]:

```
# Augmented Dickey-Fuller test on volume of google and microsoft stocks

adf = adfuller(microsoft["Volume"])
print("p-value of microsoft: {}".format(float(adf[1])))
adf = adfuller(google["Volume"])
print("p-value of google: {}".format(float(adf[1])))
```

```
p-value of microsoft: 0.0003201525277652073
p-value of google: 6.510719605768848e-07
```

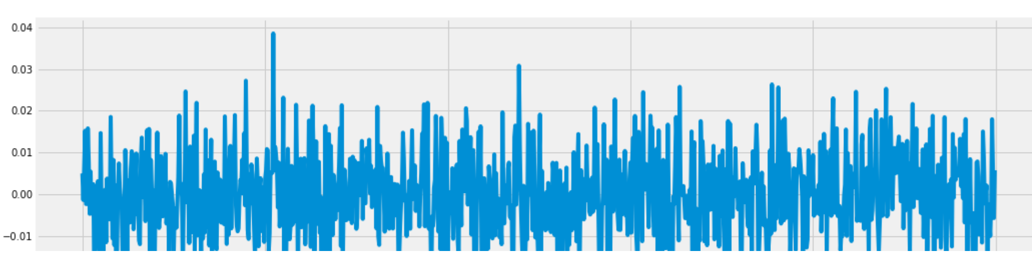
As microsoft has p-value 0.0003201525 which is less than 0.05, null hypothesis is rejected and this is not a random walk.

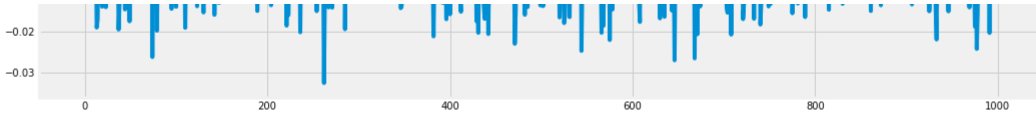
Now google has p-value 0.0000006510 which is more than 0.05, null hypothesis is rejected and this is not a random walk.

Generating a random walk

In [51]:

```
seed(42)
rcParams['figure.figsize'] = 16, 6
random_walk = normal(loc=0, scale=0.01, size=1000)
plt.plot(random_walk)
plt.show()
```





In [52]:

```
fig = ff.create_distplot([random_walk],['Random Walk'],bin_size=0.001)
iplot(fig, filename='Basic Distplot')
```

3.4 Stationarity

A stationary time series is one whose statistical properties such as mean, variance, autocorrelation, etc. are all constant over time.

- Strong stationarity: is a stochastic process whose unconditional joint probability distribution does not change when shifted in time. Consequently, parameters such as mean and variance also do not change over time.
- Weak stationarity: is a process where mean, variance, autocorrelation are constant throughout the time

Stationarity is important as non-stationary series that depend on time have too many parameters to account for when modelling the time series. diff() method can easily convert a non-stationary series to a stationary series.

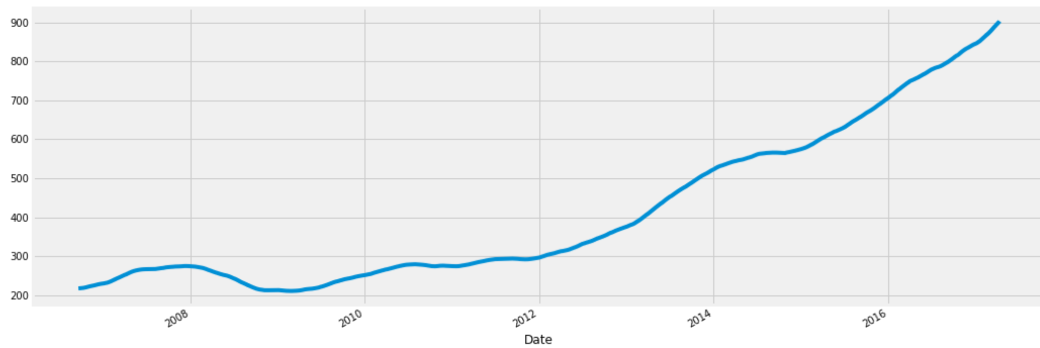
We will try to decompose seasonal component of the above decomposed time series.

In [53]:

```
# The original non-stationary plot
decomposed_google_volume.trend.plot()
```

Out[53]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f3da03615f8>
```

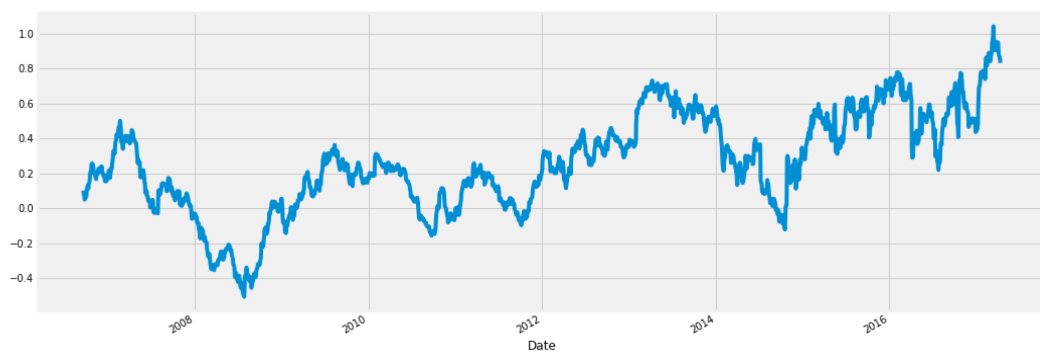


In [54]:

```
# The new stationary plot
decomposed_google_volume.trend.diff().plot()
```

Out[54]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f3da03671d0>
```



4. Modelling using statstools

4.1 AR models

An autoregressive (AR) model is a representation of a type of random process; as such, it is used to describe certain time-varying processes in nature, economics, etc. The autoregressive model specifies that the output variable depends linearly on its own previous values and on a stochastic term (an imperfectly predictable term); thus the model is in the form of a stochastic difference equation.

AR(1) model

$R_t = \mu + \phi R_{t-1} + \varepsilon_t$

$R_t = \mu + \phi R_{t-1} + \varepsilon_t$

As RHS has only one lagged value(R_{t-1})this is called AR model of order 1 where μ is mean and ε is noise at time t

If ϕ = 1, it is random walk. Else if ϕ = 0, it is white noise. Else if -1 < ϕ < 1, it is stationary. If ϕ is -ve, there is men reversion. If ϕ is +ve, there is momentum.

AR(2) model

$R_t = \mu + \phi_1 R_{t-1} + \phi_2 R_{t-2} + \varepsilon_t$

AR(3) model

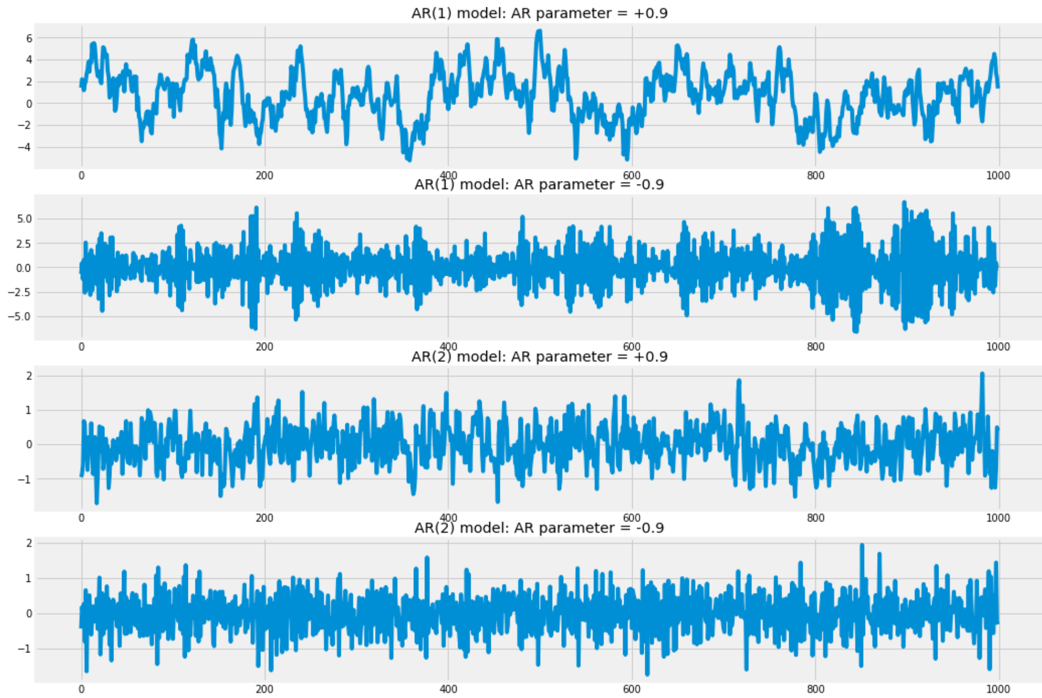
$R_t = \mu + \phi_1 R_{t-1} + \phi_2 R_{t-2} + \phi_3 R_{t-3} + \varepsilon_t$

Simulating AR(1) model

In [55]:

```
# AR(1) MA(1) model:AR parameter = +0.9
rcParams['figure.figsize'] = 16, 12
plt.subplot(4,1,1)
ar1 = np.array([1, -0.9]) # We choose -0.9 as AR parameter is +0.9
ma1 = np.array([1])
AR1 = ArmaProcess(ar1, ma1)
sim1 = AR1.generate_sample(nsample=1000)
plt.title('AR(1) model: AR parameter = +0.9')
plt.plot(sim1)
# We will take care of MA model later
# AR(1) MA(1) AR parameter = -0.9
plt.subplot(4,1,2)
ar2 = np.array([1, 0.9]) # We choose +0.9 as AR parameter is -0.9
ma2 = np.array([1])
AR2 = ArmaProcess(ar2, ma2)
sim2 = AR2.generate_sample(nsample=1000)
plt.title('AR(1) model: AR parameter = -0.9')
plt.plot(sim2)
# AR(2) MA(1) AR parameter = 0.9
plt.subplot(4,1,3)
ar3 = np.array([2, -0.9]) # We choose -0.9 as AR parameter is +0.9
ma3 = np.array([1])
AR3 = ArmaProcess(ar3, ma3)
sim3 = AR3.generate_sample(nsample=1000)
plt.title('AR(2) model: AR parameter = +0.9')
plt.plot(sim3)
# AR(2) MA(1) AR parameter = -0.9
plt.subplot(4,1,4)
ar4 = np.array([2, 0.9]) # We choose +0.9 as AR parameter is -0.9
ma4 = np.array([1])
AR4 = ArmaProcess(ar4, ma4)
sim4 = AR4.generate_sample(nsample=1000)
```

```
plt.title('AR(2) model: AR parameter = -0.9')
plt.plot(sim4)
plt.show()
```



Forecasting a simulated model

In [56]:

```
model = ARMA(sim1, order=(1,0))
result = model.fit()
print(result.summary())
print("μ={} ,φ={}".format(result.params[0],result.params[1]))
```

```
                              ARMA Model Results

==============================================================================
=========
Dep. Variable:                      y   No. Observations:
     1000
Model:                     ARMA(1, 0)   Log Likelihood
-1415.701
Method:                       css-mle   S.D. of innovations
    0.996
Date:                Thu, 02 Aug 2018   AIC
2837.403
Time:                        14:43:19   BIC
2852.126
Sample:                             0   HQIC
2842.998
```

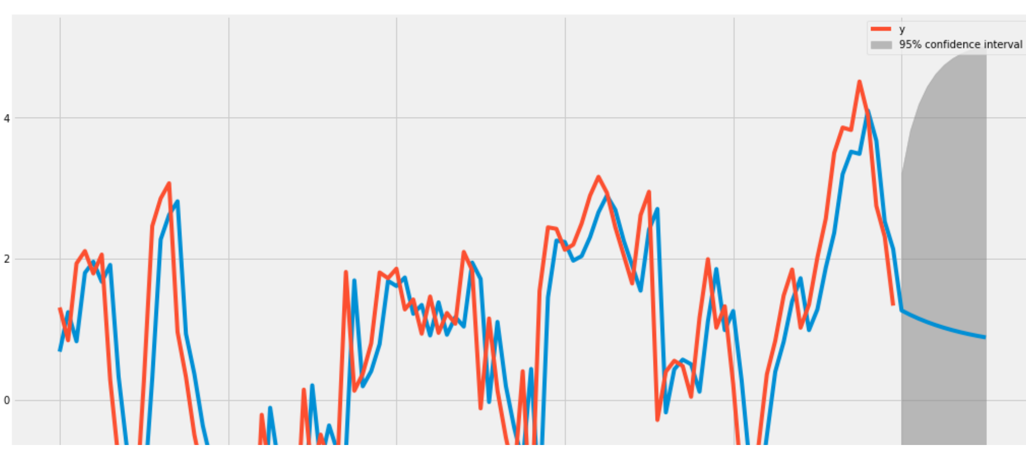
```
==============================================================================
=========
                 coef    std err          z      P>|z|      [0.025
   0.975]
------------------------------------------------------------------------------
---------
const          0.7072      0.288      2.454      0.014       0.142
    1.272
ar.L1.y        0.8916      0.014     62.742      0.000       0.864
    0.919
                                    Roots

==============================================================================
========
                 Real          Imaginary           Modulus         F
requency
------------------------------------------------------------------------------
--------
AR.1            1.1216          +0.0000j            1.1216
  0.0000
------------------------------------------------------------------------------
--------
μ=0.7072025170552714 ,φ=0.8915815634822984
```

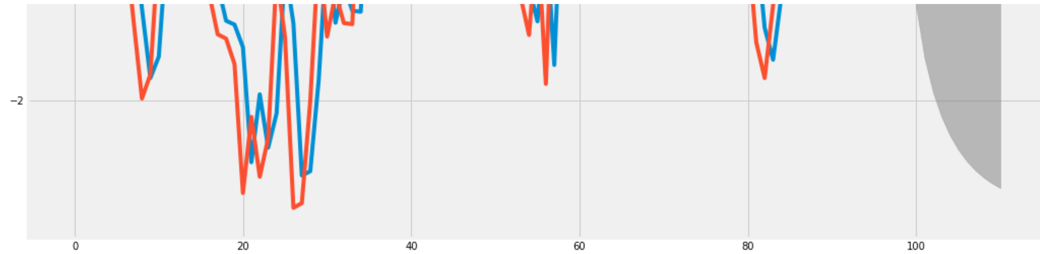
ϕ is around 0.9 which is what we chose as AR parameter in our first simulated model.

Predicting the models

In [57]:

```
# Predicting simulated AR(1) model
result.plot_predict(start=900, end=1010)
plt.show()
```



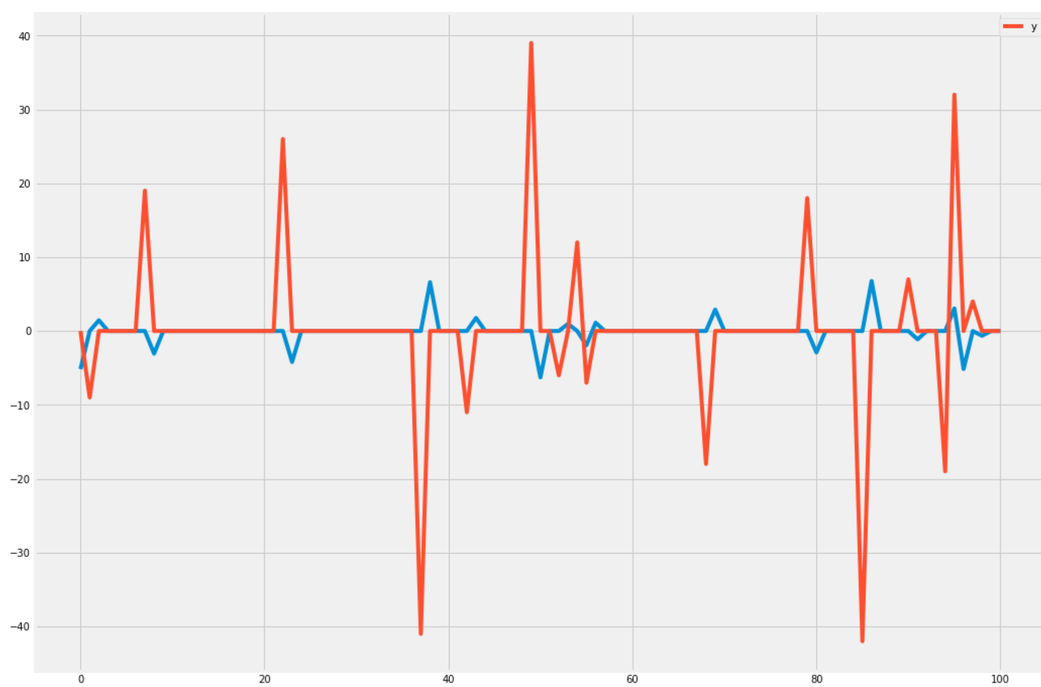


In [58]:
```
rmse = math.sqrt(mean_squared_error(sim1[900:1011], result.predict(sta
rt=900,end=999)))
print("The root mean squared error is {}.".format(rmse))
```

```
The root mean squared error is 1.0408054544358292.
```

y is predicted plot. Quite neat!

In [59]:
```
# Predicting humidity level of Montreal
humid = ARMA(humidity["Montreal"].diff().iloc[1:].values, order=(1,0))
res = humid.fit()
res.plot_predict(start=1000, end=1100)
plt.show()
```



In [60]:
```
rmse = math.sqrt(mean_squared_error(humidity["Montreal"].diff().iloc[9
00:1000].values, result.predict(start=900,end=999)))
```

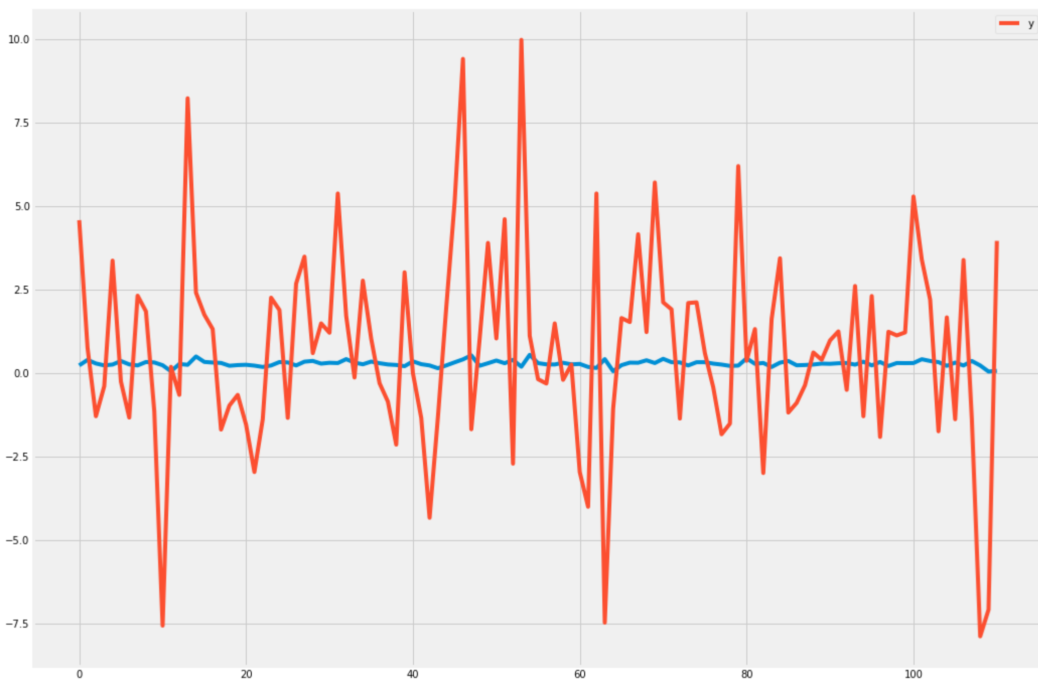
```
print("The root mean squared error is {}.".format(rmse))
```

```
The root mean squared error is 7.218388589479766.
```

Not quite impressive. But let's try google stocks.

In [61]:

```
# Predicting closing prices of google
humid = ARMA(google["Close"].diff().iloc[1:].values, order=(1,0))
res = humid.fit()
res.plot_predict(start=900, end=1010)
plt.show()
```



There are always better models.

4.2 MA models

The moving-average (MA) model is a common approach for modeling univariate time series. The moving-average model specifies that the output variable depends linearly on the current and various past values of a stochastic (imperfectly predictable) term.

MA(1) model

$R_t = \mu + \epsilon_t 1 + \theta\epsilon_{t-1}$

It translates to Today's returns = mean + today's noise + yesterday's noise

As there is only 1 lagged value in RHS, it is an MA model of order 1

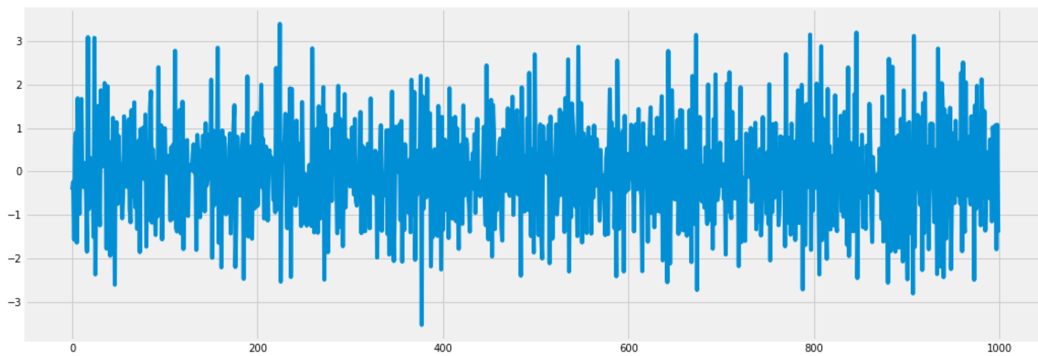
Simulating MA(1) model

In [62]:

```
rcParams['figure.figsize'] = 16, 6
ar1 = np.array([1])
ma1 = np.array([1, -0.5])
MA1 = ArmaProcess(ar1, ma1)
sim1 = MA1.generate_sample(nsample=1000)
plt.plot(sim1)
```

Out[62]:

```
[<matplotlib.lines.Line2D at 0x7f3d9801c8d0>]
```



Forecasting the simulated MA model

In [63]:

```
model = ARMA(sim1, order=(0,1))
result = model.fit()
print(result.summary())
print("μ={} ,θ={}".format(result.params[0],result.params[1]))
```

```
                              ARMA Model Results                              

==============================================================================
=========
Dep. Variable:                      y   No. Observations:                   
     1000
Model:                     ARMA(0, 1)   Log Likelihood                     
-1423.276
Method:                       css-mle   S.D. of innovations                 
    1.004
Date:                Thu, 02 Aug 2018   AIC                                 
2852.553
Time:                        14:43:24   BIC
```

```
2867.276
Sample:                          0   HQIC
2858.148


=====================================================================
=========
                 coef    std err          z      P>|z|      [0.025
   0.975]
---------------------------------------------------------------------
---------
const          -0.0228      0.014     -1.652      0.099      -0.050
    0.004
ma.L1.y        -0.5650      0.027    -20.797      0.000      -0.618
   -0.512
                                    Roots

=====================================================================
========
                 Real          Imaginary           Modulus         F
requency
---------------------------------------------------------------------
--------
MA.1            1.7699           +0.0000j            1.7699
  0.0000
---------------------------------------------------------------------
--------
μ=-0.022847169009088644 ,θ=-0.5650012298416457
```

Prediction using MA models

In [64]:

```
# Forecasting and predicting montreal humidity
model = ARMA(humidity["Montreal"].diff().iloc[1:].values, order=(0,3))
result = model.fit()
print(result.summary())
print("μ={} ,θ={}".format(result.params[0],result.params[1]))
result.plot_predict(start=1000, end=1100)
plt.show()
```

```
                              ARMA Model Results

=====================================================================
=========
Dep. Variable:                       y   No. Observations:
    45251
Model:                      ARMA(0, 3)   Log Likelihood              -1
```

```
53516.982
Method:                       css-mle   S.D. of innovations
    7.197
Date:                Thu, 02 Aug 2018   AIC                            3
07043.965
Time:                        14:43:32   BIC                            3
07087.564
Sample:                             0   HQIC                           3
07057.686


======================================================================
=========
                 coef    std err          z      P>|z|      [0.025
   0.975]
----------------------------------------------------------------------
---------
const         -0.0008      0.031     -0.025      0.980      -0.061
    0.060
ma.L1.y       -0.1621      0.005    -34.507      0.000      -0.171
   -0.153
ma.L2.y        0.0386      0.005      8.316      0.000       0.030
    0.048
ma.L3.y        0.0357      0.005      7.446      0.000       0.026
    0.045
                                    Roots

======================================================================
========
                 Real          Imaginary           Modulus         F
requency
----------------------------------------------------------------------
--------
MA.1            1.4520           -2.2191j            2.6519
-0.1578
MA.2            1.4520           +2.2191j            2.6519
  0.1578
MA.3           -3.9867           -0.0000j            3.9867
-0.5000
----------------------------------------------------------------------
--------
μ=-0.0007772680242180366 ,θ=-0.16209499431431182
```

In [65]:
```
rmse = math.sqrt(mean_squared_error(humidity["Montreal"].diff().iloc[1
000:1101].values, result.predict(start=1000,end=1100)))
print("The root mean squared error is {}.".format(rmse))
```

```
The root mean squared error is 11.345129665763626.
```

Now, for ARMA models.

4.3 ARMA models

Autoregressive–moving-average (ARMA) models provide a parsimonious description of a (weakly) stationary stochastic process in terms of two polynomials, one for the autoregression and the second for the moving average. It's the fusion of AR and MA models.

ARMA(1,1) model

$R_t = \mu + \phi R_{t-1} + \epsilon_t + \theta\epsilon_{t-1}$

Basically, Today's return = mean + Yesterday's return + noise + yesterday's noise.

Prediction using ARMA models

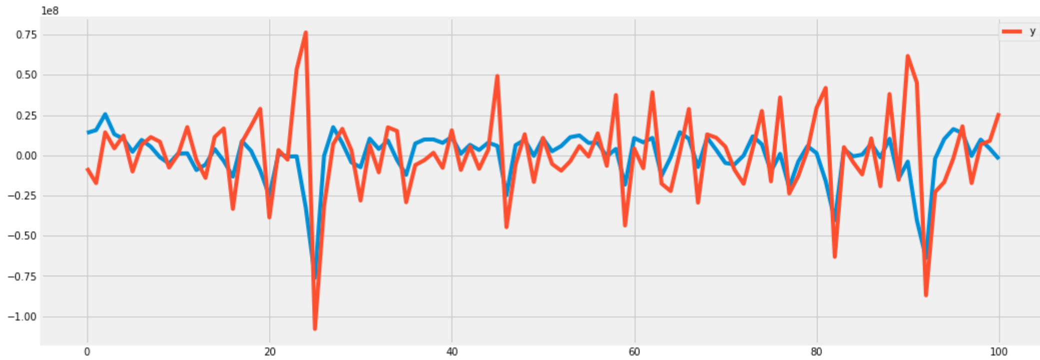
I am not simulating any model because it's quite similar to AR and MA models. Just forecasting and predictions for this one.

In [66]:
```
# Forecasting and predicting microsoft stocks volume
model = ARMA(microsoft["Volume"].diff().iloc[1:].values, order=(3,3))
result = model.fit()
print(result.summary())
print("μ={}, φ={}, θ={}".format(result.params[0],result.params[1],resu
lt.params[2]))
result.plot_predict(start=1000, end=1100)
plt.show()
```

```
                              ARMA Model Results
==============================================================================
=========
Dep. Variable:                      y   No. Observations:
     3018
```

```
Model:                     ARMA(3, 3)   Log Likelihood              -
55408.974
Method:                       css-mle   S.D. of innovations       227
51607.792
Date:                Thu, 02 Aug 2018   AIC                         1
10833.948
Time:                        14:43:52   BIC                         1
10882.047
Sample:                             0   HQIC                        1
10851.244


=====================================================================
=========
                 coef    std err          z      P>|z|      [0.025
   0.975]
---------------------------------------------------------------------
---------
const       -2.03e+04   9914.912     -2.047      0.041   -3.97e+04
-864.350
ar.L1.y        0.2053      0.160      1.287      0.198      -0.107
    0.518
ar.L2.y        0.7297      0.179      4.080      0.000       0.379
    1.080
ar.L3.y       -0.1413      0.057     -2.467      0.014      -0.254
   -0.029
ma.L1.y       -0.8117      0.157     -5.165      0.000      -1.120
   -0.504
ma.L2.y       -0.7692      0.258     -2.978      0.003      -1.275
   -0.263
ma.L3.y        0.5853      0.130      4.494      0.000       0.330
    0.841
                                    Roots

=====================================================================
========
                 Real          Imaginary           Modulus         F
requency
---------------------------------------------------------------------
--------
AR.1          -1.1772           +0.0000j            1.1772
  0.5000
AR.2           1.1604           +0.0000j            1.1604
  0.0000
AR.3           5.1820           +0.0000j            5.1820
  0.0000
MA.1          -1.1579           +0.0000j            1.1579
  0.5000
MA.2           1.0075           +0.0000j            1.0075
  0.0000
```

```
MA.3            1.4647          +0.0000j            1.4647
  0.0000
-----------------------------------------------------------------------
--------
μ=-20297.220687001212, φ=0.2053008905725337, θ=0.7296681716172797
```



In [67]:

```
rmse = math.sqrt(mean_squared_error(microsoft["Volume"].diff().iloc[10
00:1101].values, result.predict(start=1000,end=1100)))
print("The root mean squared error is {}.".format(rmse))
```

```
The root mean squared error is 38038241.66905847.
```

ARMA model shows much better results than AR and MA models.

4.4 ARIMA models

An autoregressive integrated moving average (ARIMA) model is a generalization of an autoregressive moving average (ARMA) model. Both of these models are fitted to time series data either to better understand the data or to predict future points in the series (forecasting). ARIMA models are applied in some cases where data show evidence of non-stationarity, where an initial differencing step (corresponding to the "integrated" part of the model) can be applied one or more times to eliminate the non-stationarity. ARIMA model is of the form: ARIMA(p,d,q): p is AR parameter, d is differential parameter, q is MA parameter

ARIMA(1,0,0)

$\mathbf{y_t = a_1 y_{t-1} + \epsilon_t}$

ARIMA(1,0,1)

$\mathbf{y_t = a_1 y_{t-1} + \epsilon_t + b_1 \epsilon_{t-1}}$

ARIMA(1,1,1)

$\mathbf{\Delta y_t = a_1 \Delta y_{t-1} + \epsilon_t + b_1 \epsilon_{t-1}}$ where $\mathbf{\Delta y_t = y_t - y_{t-1}}$

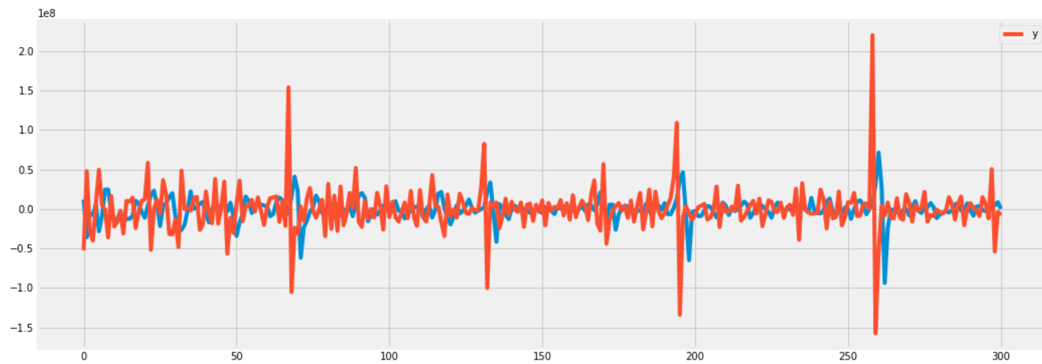
Prediction using ARIMA model

In [68]:

```
# Predicting the microsoft stocks volume
rcParams['figure.figsize'] = 16, 6
model = ARIMA(microsoft["Volume"].diff().iloc[1:].values, order=(2,1,0
))
result = model.fit()
print(result.summary())
result.plot_predict(start=700, end=1000)
plt.show()
```

```
                              ARIMA Model Results

==============================================================================
=========
Dep. Variable:                    D.y   No. Observations:
     3017
Model:                 ARIMA(2, 1, 0)   Log Likelihood                  -
56385.467
Method:                       css-mle   S.D. of innovations       316
47215.014
Date:                Thu, 02 Aug 2018   AIC                             1
12778.933
Time:                        14:44:02   BIC                             1
12802.981
Sample:                             1   HQIC                            1
12787.581


==============================================================================
=========
                 coef    std err          z      P>|z|      [0.025
   0.975]
------------------------------------------------------------------------------
---------
const       9984.0302   2.48e+05      0.040      0.968   -4.75e+05
4.95e+05
ar.L1.D.y     -0.8716      0.016    -53.758      0.000      -0.903
   -0.840
ar.L2.D.y     -0.4551      0.016    -28.071      0.000      -0.487
   -0.423
                                    Roots

==============================================================================
========
                 Real          Imaginary           Modulus         F
```

```
requency
-----------------------------------------------------------------------------
--------
AR.1            -0.9575            -1.1315j             1.4823
-0.3618
AR.2            -0.9575            +1.1315j             1.4823
  0.3618
-----------------------------------------------------------------------------
--------
```



In [69]:

```
rmse = math.sqrt(mean_squared_error(microsoft["Volume"].diff().iloc[70
0:1001].values, result.predict(start=700,end=1000)))
print("The root mean squared error is {}.".format(rmse))
```

```
The root mean squared error is 61937593.98493614.
```

Taking the slight lag into account, this is a fine model.

4.5 VAR models

Vector autoregression (VAR) is a stochastic process model used to capture the linear interdependencies among multiple time series. VAR models generalize the univariate autoregressive model (AR model) by allowing for more than one evolving variable. All variables in a VAR enter the model in the same way: each variable has an equation explaining its evolution based on its own lagged values, the lagged values of the other model variables, and an error term. VAR modeling does not require as much knowledge about the forces influencing a variable as do structural models with simultaneous equations: The only prior knowledge required is a list of variables which can be hypothesized to affect each other intertemporally.

A two-series, VAR(1) Model:

$$y_{1,t} = c_1 + \phi_{11,1}y_{1,t-1} + \phi_{12,1}y_{2,t-1} + u_{1,t}$$
$$y_{2,t} = c_2 + \phi_{21,1}y_{1,t-1} + \phi_{22,1}y_{2,t-1} + u_{2,t}$$

where

- $\epsilon_{1,t}$ and $\epsilon_{2,t}$ are white noise processes that may be contemporaneously correlated.
- $\phi_{ii,l}$ captures the effect of the l^{th} lag of series y_i on itself
- $\phi_{ij,l}$ captures the effect of the l^{th} lag of series y_j on y_i

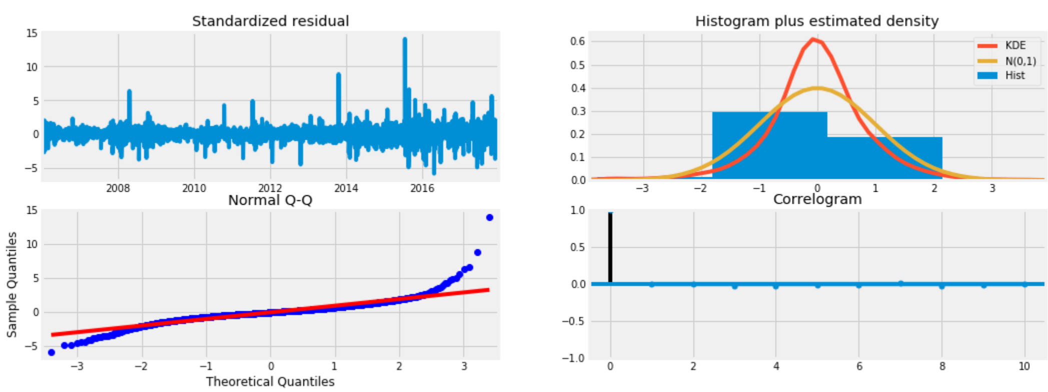
In [70]:

```
# Predicting closing price of Google and microsoft
train_sample = pd.concat([google["Close"].diff().iloc[1:],microsoft["C
lose"].diff().iloc[1:]],axis=1)
model = sm.tsa.VARMAX(train_sample,order=(2,1),trend='c')
result = model.fit(maxiter=1000,disp=False)
print(result.summary())
predicted_result = result.predict(start=0, end=1000)
result.plot_diagnostics()
# calculating error
rmse = math.sqrt(mean_squared_error(train_sample.iloc[1:1002].values,
predicted_result.values))
print("The root mean squared error is {}.".format(rmse))
```

```
                           Statespace Model Results
======================================================================
=========
Dep. Variable:     ['Close', 'Close']   No. Observations:
     3018
Model:                      VARMA(2,1)   Log Likelihood             -
12185.169
                          + intercept   AIC
24404.337
Date:                 Thu, 02 Aug 2018   BIC
24506.547
Time:                         14:44:26   HQIC
24441.091
Sample:                     01-04-2006

                          - 12-29-2017

Covariance Type:                   opg

======================================================================
```

```
==============
Ljung-Box (Q):                77.92, 70.40   Jarque-Bera (JB):   5573
1.67, 15359.82
Prob(Q):                        0.00, 0.00   Prob(JB):
    0.00, 0.00
Heteroskedasticity (H):         3.35, 1.82   Skew:
    1.24, 0.29
Prob(H) (two-sided):            0.00, 0.00   Kurtosis:
  23.91, 14.04
                           Results for equation Close

======================================================================
==========
                  coef    std err          z      P>|z|      [0.025
    0.975]
----------------------------------------------------------------------
----------
const           0.2983      0.260      1.146      0.252      -0.212
     0.809
L1.Close       -0.1510      0.555     -0.272      0.785      -1.238
     0.936
L1.Close       -0.1981      5.026     -0.039      0.969     -10.049
     9.653
L2.Close        0.0028      0.037      0.075      0.940      -0.070
     0.075
L2.Close        0.3825      0.433      0.882      0.378      -0.467
     1.232
L1.e(Close)     0.1906      0.555      0.343      0.731      -0.898
     1.279
L1.e(Close)    -0.0530      5.047     -0.010      0.992      -9.946
     9.840
                           Results for equation Close

======================================================================
==========
                  coef    std err          z      P>|z|      [0.025
    0.975]
----------------------------------------------------------------------
----------
const           0.0169      0.027      0.635      0.526      -0.035
     0.069
L1.Close        0.0430      0.057      0.751      0.453      -0.069
     0.155
L1.Close       -0.4625      0.523     -0.885      0.376      -1.487
     0.562
L2.Close        0.0012      0.004      0.331      0.740      -0.006
     0.008
L2.Close       -0.0432      0.042     -1.030      0.303      -0.125
     0.039
L1.e(Close)    -0.0407      0.057     -0.713      0.476      -0.153
     0.071
```

```
          0.071
L1.e(Close)      0.4172       0.523       0.797       0.426       -0.609
          1.443
                                Error covariance matrix

======================================================================
===================
                           coef    std err          z      P>|z|
[0.025      0.975]
----------------------------------------------------------------------
-------------------
sqrt.var.Close          6.9023      0.041    167.093      0.000
  6.821       6.983
sqrt.cov.Close.Close    0.2926      0.005     57.547      0.000
  0.283       0.303
sqrt.var.Close          0.4809      0.003    163.032      0.000
  0.475       0.487
======================================================================
===================

Warnings:
[1] Covariance matrix calculated using the outer product of gradients
(complex-step).
The root mean squared error is 3.674416216782386.
```



4.6 State Space methods

A general state space model is of the form

$y_t=Z_t\alpha_t+d_t+\varepsilon_t$

$\alpha_t=T_t\alpha_t-1+c_t+R_t\eta_t$

where y_t refers to the observation vector at time t, α_t refers to the (unobserved) state vector at time t, and where the irregular components are defined as

$\varepsilon_t\sim N(0,H_t)$

$\eta_t\sim N(0,Q_t)$

The remaining variables (Z_t,d_t,H_t,T_t,c_t,R_t,Q_t) in the equations are matrices describing the process. Their variable names and dimensions are as follows

Z : design (k_endog×k_states×nobs)

d : obs_intercept (k_endog×nobs)

H : obs_cov (k_endog×k_endog×nobs)

T : transition (k_states×k_states×nobs)

c : state_intercept (k_states×nobs)

R : selection (k_states×k_posdef×nobs)

Q : state_cov (k_posdef×k_posdef×nobs)

In the case that one of the matrices is time-invariant (so that, for example, Z_t=Z_t+1 ∀ $_t$), its last dimension may be of size 1 rather than size nobs.

This generic form encapsulates many of the most popular linear time series models (see below) and is very flexible, allowing estimation with missing observations, forecasting, impulse response functions, and much more.

Source: statsmodels (https://www.statsmodels.org/dev/statespace.html)

4.6.1 SARIMA models

SARIMA models are useful for modeling seasonal time series, in which the mean and other statistics for a given season are not stationary across the years. The SARIMA model defined constitutes a straightforward extension of the nonseasonal autoregressive-moving average (ARMA) and autoregressive integrated moving average (ARIMA) models presented

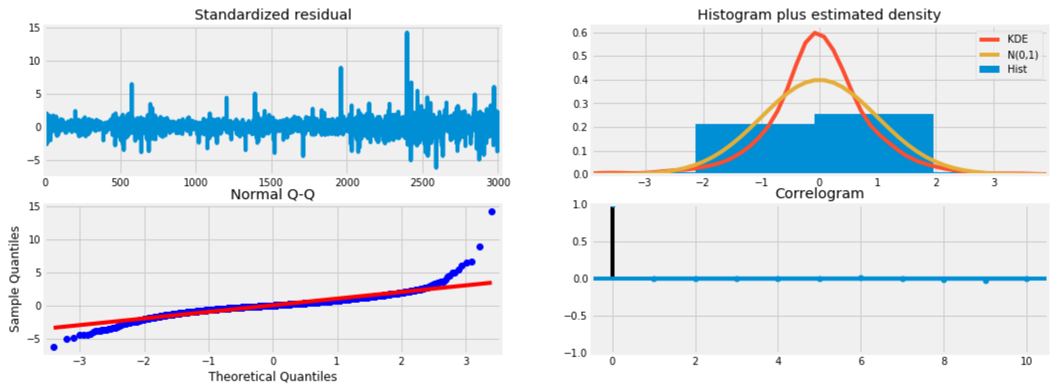
In [71]:

```
# Predicting closing price of Google'
train_sample = google["Close"].diff().iloc[1:].values
model = sm.tsa.SARIMAX(train_sample,order=(4,0,4),trend='c')
result = model.fit(maxiter=1000,disp=False)
print(result.summary())
predicted_result = result.predict(start=0, end=500)
result.plot_diagnostics()
# calculating error
rmse = math.sqrt(mean_squared_error(train_sample[1:502], predicted_res
ult))
print("The root mean squared error is {}.".format(rmse))
```

```
                           Statespace Model Results

==============================================================================
=========
Dep. Variable:                          y   No. Observations:
```

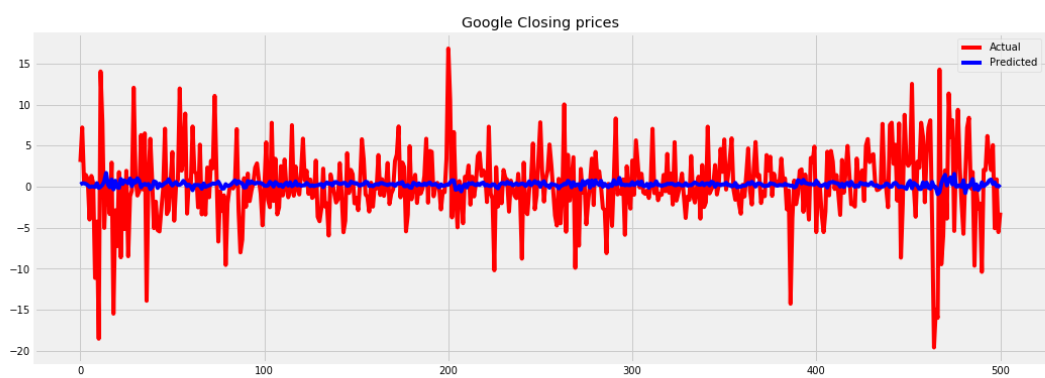
```
Dep. Variable:                      y   No. Observations:
    3018
Model:               SARIMAX(4, 0, 4)   Log Likelihood              -
10109.282
Date:                Thu, 02 Aug 2018   AIC
20238.563
Time:                        14:44:46   BIC
20298.687
Sample:                             0   HQIC
20260.183
                                 - 3018

Covariance Type:                  opg

======================================================================
=========
                 coef    std err          z      P>|z|      [0.025
   0.975]
----------------------------------------------------------------------
---------
intercept      0.2140      0.153      1.399      0.162      -0.086
    0.514
ar.L1         -0.5034      0.232     -2.172      0.030      -0.958
   -0.049
ar.L2          0.5397      0.204      2.644      0.008       0.140
    0.940
ar.L3          0.4529      0.225      2.016      0.044       0.013
    0.893
ar.L4         -0.2606      0.225     -1.160      0.246      -0.701
    0.180
ma.L1          0.5315      0.236      2.254      0.024       0.069
    0.994
ma.L2         -0.5132      0.207     -2.484      0.013      -0.918
   -0.108
ma.L3         -0.4986      0.228     -2.188      0.029      -0.945
   -0.052
ma.L4          0.2029      0.231      0.877      0.381      -0.251
    0.656
sigma2        47.5301      0.423    112.357      0.000      46.701
   48.359
======================================================================
==============
Ljung-Box (Q):                      66.54   Jarque-Bera (JB):
      50700.74
Prob(Q):                             0.01   Prob(JB):
          0.00
Heteroskedasticity (H):              3.33   Skew:
          1.16
Prob(H) (two-sided):                 0.00   Kurtosis:
         22.94
======================================================================
```

```
==============

Warnings:
[1] Covariance matrix calculated using the outer product of gradients (complex-step).
The root mean squared error is 4.370002341096441.
```



In [72]:

```
plt.plot(train_sample[1:502],color='red')
plt.plot(predicted_result,color='blue')
plt.legend(['Actual','Predicted'])
plt.title('Google Closing prices')
plt.show()
```



4.6.2 Unobserved components

A UCM decomposes the response series into components such as trend, seasons, cycles, and the regression effects due to predictor series. The following model shows a possible scenario:

$$
\begin{aligned}
y_t &= \mu_t + \gamma_t + \psi_t + \sum_{j=1}^{m} \beta_j x_{jt} + \epsilon_t \\
\epsilon_t &\sim i.i.d.\ N(0, \sigma_\epsilon^2)
\end{aligned}
$$

Source:
http://support.sas.com/documentation/cdl/en/etsug/66840/HTML/default/viewer.htm#etsug_ucm_deta
(http://support.sas.com/documentation/cdl/en/etsug/66840/HTML/default/viewer.htm#etsug_ucm_det

In [73]:

```
# Predicting closing price of Google'
train_sample = google["Close"].diff().iloc[1:].values
model = sm.tsa.UnobservedComponents(train_sample,'local level')
result = model.fit(maxiter=1000,disp=False)
print(result.summary())
predicted_result = result.predict(start=0, end=500)
result.plot_diagnostics()
# calculating error
rmse = math.sqrt(mean_squared_error(train_sample[1:502], predicted_res
ult))
print("The root mean squared error is {}.".format(rmse))
```

```
                        Unobserved Components Results

======================================================================
=========
Dep. Variable:                         y   No. Observations:
     3018
Model:                       local level   Log Likelihood              -
10116.511
Date:                   Thu, 02 Aug 2018   AIC
20237.023
Time:                           14:44:48   BIC
20249.047
Sample:                                0   HQIC
20241.347
                                  - 3018

Covariance Type:                     opg
```


Comments (51)

All Comments

Sort by Hotness

Click here to enter a comment...

Amardeep Chauh... • Posted on Latest Version • 10 days ago • Options • Reply

1

One of the great work I am going through. Thanks for sharing.

Abdullah Çakmak • Posted on Latest Version • a month ago • Options • Reply 1

Thank you for sharing.

szrlee • Posted on Latest Version • a month ago • Options • Reply 1

Thank you for your contributions!

AishwaryaSingh • Posted on Latest Version • a month ago • Options • Reply 1

Really appreciate the efforts you put in! Great job!! Could you explain more about the error calculation for VAR?

Mayur Kala • Posted on Latest Version • 2 months ago • Options • Reply 1

Thanks Siddharth,

Your work here with above example helped me a lot with time series. This consolidates most of the manipulation in Time series and best for a beginner.

I am struggling with these two concepts in Time Series: 1) Checking for the stationary time series for Stock Market. 2) Performing the Prediction and Forecasting of Stock Prices.

If you get time so please extend these topics in above notebook.

Regards, Mayur

> **Siddharth Yad...** Kernel Author • Posted on Latest Version • 2 months ago • Options • Reply 0
>
> As a matter of fact, I have a kernel that deals with time series prediction of stock prices. Here's the link: https://www.kaggle.com/thebrownviking20/intro-to-recurrent-neural-networks-lstm-gru

Leonardo Ferreira • Posted on Latest Version • 3 months ago • Options • Reply 1

Congratulations bro, excellent work. I learned a lot

Xavier • Posted on Version 17 • 3 months ago • Options • Reply 1

Amazing kernel Siddharth!!

lpkmath • Posted on Version 17 • 3 months ago • Options • Reply 1

Extremely helpful!! Great reference!!

Nick Brooks • Posted on Version 12 • 3 months ago • Options • Reply 1

Very comprehensive! Will definitely reference this.

Prakhar Gupta • Posted on Latest Version • 3 months ago • Options • Reply 2

Nice work I am thinking to make a equivalent R code in rmd so other can refer to our codes when need. Do i have permission to use your data?

Siddharth Yad... Kernel Author • Posted on Latest Version • 3 months ago • Options • Reply 0

Yes, you can. Notify me when you put it on kaggle. As I am learning R, it will be very knowledgeable for me.

Krishna • Posted on Version 12 • 3 months ago • Options • Reply 1

Hi, I'm new to time series analysis. Is it sufficient? If we have date and time in our dataset to do time series analysis

Siddharth Yad... Kernel Author • Posted on Version 12 • 3 months ago • Options • Reply 0

You should not just consume information for data science from just one place. Refer to at least 5 sources. This project is a cumulative result of 5-6 sources and I am yet to add more material. Even though I am trying to add as much as I can, there's a chance I would leave something behind. Stay tuned for updates. Till then go to `time series` and `time series analysis` tags through search bar for more sources.

Gabriel Preda • Posted on Version 12 • 3 months ago • Options • Reply 1

Good work. May I suggest to extend this to a training material?

Siddharth Yad... Kernel Author • Posted on Version 12 • 3 months ago • Options • Reply 0

Cool sure!

Chirag Choudhary • Posted on Version 11 • 3 months ago • Options • Reply 1

Impressive!!

morenoh149 • Posted on Version 11 • 3 months ago • Options • Reply 1

shouldn't the description for `Weak Stationarity` read `mean, variance, autocorrelation are time variant` , e.g., the mean may vary due to the time of year?

Siddharth Yad... Kernel Author • Posted on Version 11 • 3 months ago • Options • Reply 0

time invariant means that mean, variance and autocorrelation are constant throughout the time.

morenoh149 • Posted on Version 12 • 3 months ago • Options • Reply 0

yes. `Weak Stationarity` means the summary statistic varies through time right?

Siddharth Yad... Kernel Author • Posted on Version 12 • 3 months ago • Options • Reply 0

Yeah.

kairos • Posted on Version 11 • 3 months ago • Options • Reply 1

Great work!

Michal Haltuf • Posted on Version 11 • 3 months ago • Options • Reply 1

> p-value of google: 6.51071960576848e-07 Now google has p-value 6.5107 which is more than 0.05, null hypothesis is accept and this is a random walk.¶

I believe this is not correct ... 6.5107e-07 is 0,00000065107 which is less than 0.05

Siddharth Yad... Kernel Author • Posted on Version 12 • 3 months ago • Options • Reply 0

First time I saw 6.5107, I had a doubt but I didn't see "e" that time. I didn't realize it was a scientific notation not decimal. You are right. Thanks for pointing this out. I have rectified this in the latest version.

bizolus • Posted on Version 11 • 3 months ago • Options • Reply 1

Awesome work. Very educative and a refresher on techniques I learnt during my time series course

Siddharth Yad... Kernel Author • Posted on Version 11 • 3 months ago • Options • Reply 0

Thanks! If you like my content, please upvote.

ToddPappas • Posted on Version 11 • 3 months ago • Options • Reply 1

Great Work! Lot's of work!

Michashak • Posted on Version 11 • 3 months ago • Options • Reply 1

Thanks. Great job.

Janio Alexander B... • Posted on Version 8 • 4 months ago • Options • Reply 1

Siddharth! Really good work! Thanks for sharing, I will definitely use this as a reference for my Time Series Forecasting Project!

Siddharth Yad... Kernel Author • Posted on Version 8 • 4 months ago • Options • Reply 1

This is just the beginning. I am going to add more great content. Just open this kernel everytime you see it. Thanks for your comment.

Janio Alexande... • Posted on Version 8 • 4 months ago • Options • Reply 1

Hey again Siddharth, If my dates are daily and according to my dataset the datatype of date is datetime64, how can I change its frequency from daily to monthly. My main goal is to capture certain seasonality trends but I have really struggle changing my dates from daily to monthly. Thanks for the help.

Siddharth Yad... Kernel Author • Posted on Version 8 • 4 months ago • Options • Reply 1

Check Section 1.8 Resampling.

Janio Alexande... • Posted on Version 11 • 3 months ago • Options • Reply 3

Incredible updates! One questions Siddharth, did you had to pay for the Datacamp courses?

Siddharth Yad... Kernel Author • Posted on Version 11 • 3 months ago • Options • Reply 0

No, i used a coupon for first free month. BTW next update will be even better.

Somin Wadhwa • Posted on Version 1 • 4 months ago • Options • Reply 1

Excellent kernel & a very intriguing choice of topic! (Y)

Siddharth Yad... Kernel Author • Posted on Version 2 • 4 months ago • Options • Reply 0

Thanks a lot! I will add more stuff ASAP.

menglanse • Posted on Version 11 • 3 months ago • Options • Reply 0

Can you also make a list of libraries you imported?

Siddharth Yad... Kernel Author • Posted on Version 11 • 3 months ago • Options • Reply 0

The list of libraries is visible now!

bizolus • Posted on Version 11 • 3 months ago • Options • Reply 0

GSD • Posted on Version 12 • 3 months ago • Options • Reply 0

Thanks Siddharth for sharing this awesome and easy to follow kernel on time series..Found it extremely useful and bookmarked it for my reference ...As a beginner to python , I find your kernels easy to follow ..Do check out my python kernels. Keep sharing such amazing works..

Siddharth Yad... Kernel Author • Posted on Version 12 • 3 months ago • Options • Reply 0

Thanks! Kindly upvote if you like this!

Chenlin WANG • Posted on Version 12 • 3 months ago • Options • Reply 0

Waiting for your update! Recently I met a problem about doing regression analysis on multivariables, and among them there is a time serise, I'm not sure whether to treat the timestamps as a variable in the regression or extract the timestamps to make a time serise. Will you come to something like multivariables in your future update?

Siddharth Yad... Kernel Author • Posted on Version 12 • 3 months ago • Options • Reply 0

If you want to use timestamps as variables, you should do some feature engineering. Extract features like year, month, day, hour etc. from the timestamp and then drop it from data. Then perform your regression analysis.

If the sequence or time matters for the data and you want to use the timestamp as a series, you can always use RNNs with LSTMs for regression. It is much more efficient. My most recent kernel is actually a tutorial on RNNs with LSTMs where I predict stock prices from time series data of IBM stocks. Check it out here: https://www.kaggle.com/thebrownviking20/intro-to-recurrent-neural-networks-using-lstms

Chenlin WANG • Posted on Version 13 • 3 months ago • Options • Reply 0

Much obliged! I at first wanted to try quantile regression(because I set bigger punishment on prediction that is lower than the true value) so I might extract out year, month, day variables as you suggested, however during data exploring I found some season pattern in the time sequence. Now I'm quite confused about two probelms below: 1. If I use timestamps as variables, what can I do about the season patterns? Also can LSTM deal with season patterns automatically or the season pattern is something I have to separate when doing data cleansing? 2. Is it possible for LSTM to integrate with quantile analyze?

Siddharth Yad... Kernel Author • Posted on Version 14 • 3 months ago • Options • Reply 0

It doesn't matter whether the time series is stationary, LSTM perform well on any kind of sequential data(even quantile). LSTM deal with seasonal pattern auomatically. As long as there are no null values, LSTM can efficiently model any time series.

Chenlin WANG • Posted on Version 14 • 3 months ago • Options • Reply 0

Wow, thanks a lot! I'll try the LSTM, seems like a super powerful model!

Siddharth Yad... Kernel Author • Posted on Version 14 • 3 months ago • Options • Reply 1

Check out my LSTM kernel then https://www.kaggle.com/thebrownviking20/intro-to-recurrent-neural-networks-lstm-gru

michantp • Posted on Latest Version • 6 days ago • Options • Reply

0

Hey Siddharth,

I greately appreciate your kernel! Well done.

Can I ask a quick question: I am not sure I completely understand your interpretation of the AugmentedAdFuller test at 50. I thought the test applies to stationarity. If I understand this correctly, a random walk is non-stationary in general with or without drift. As far as I understand your output, your p-values for the AdFuller test mean they are stationary processes since we can reject the H_0 for both time series because the H_0 for the adfuller() is: is non-stationary. So, do I understand you correctly that non-stationarity is sufficient to conclude that both series cannot be random walks? It should be, but I just want to be sure I understand this.

Thanks!

Best, Michael

michantp • Posted on Latest Version • 5 days ago • Options • Reply

0

Hey,

not sure if that is a mistake but in 47 you name the decomposition "volume" but use the "High" column of the google data set.

Cheers

Carlo Lepelaars • Posted on Latest Version • 5 days ago • Options • Reply

0

Amazing! How are you creating the table of contents with links to certain sections in the notebook?

3 months ago

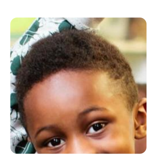
This Comment was deleted.

Similar Kernels

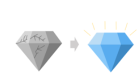
Smoothing Methods -

Introduction In Time

Time Series Prediction



DonorChoose:



Avito EDA, FE, Time

Weighted Moving Average | Series - Moving Average | Tutorial With EDA | Complete EDA + Time Series Analysis✓✓ | Series, DT Visualization ✓✓

Our Team Terms Privacy Contact/Support