Part A: Arima and Seasonal Arima

Jeremy Ames; Mustapha Farram; Rabea Radman

ARIMA and Seasonal ARIMA

Autoregressive Integrated Moving Averages The general process for ARIMA models is the following:

Get the Data Clean the Data Visualize the Time Series Data Make the time series data stationary Plot the Correlation and AutoCorrelation Charts Construct the ARIMA Model Use the model to make predictions

'''Statsmodels is a Python module that allows users to explore data,estimate statistical models, and perform statistical test. It provides classes and functions for the estimation of many different statistical models, as well as for conducting statistical tests, and statistical data exploration. An extensive list of result statistics are available for each estimator. The results are tested against existing statistical packages to ensure that they are correct. The package is released under the open source Modified BSD (3-clause) license. The online documentation is hosted at statsmodels.org'''

In [164]:
```
# Autoregressive Integrated Moving Average (ARIMA) model
```

'''It is a generalization of a an autoregressive moving (ARMA) model. Both of those models (ARIMA and ARMA)are fitted to time series data either to better understand the data or to predict future points in the series (forecasting).

ARIMA model have two types:

1- Non-seasonal ARIMA for non-seasonal Data; 2- Seasonal ARIMA for SEasonal Data

ARIMA models are applied in some cases where data show evidence of non-stationaryity'''

In [165]:
```
# Major Components of ARIMA model in general
```

1- Non-seasonal ARIMA models are generally denoteed ARIMA (pdq) where parameteter p,q and q are non-negative integers;

a) AR(p): Autoregression: A basic regression model that utilizes the dependent relationship between a current observation and observations over a previous period;

b) I(d): Integrated: Differencing observations (subtracting and observation at the previous time step) in order to make the time series stationary;

c) MA(q): Moving Average: A model that uses the dependency between an observation and a residual error from a moving average model applied to lagged observations.

In [166]:
```
# Stationary vs Non-Stationary Data
```

To effectivelly use ARIMA, we need to understand Stationarity in our data. So, what makes a data set Stationary? A Stationary series has constant mean and variance over time.

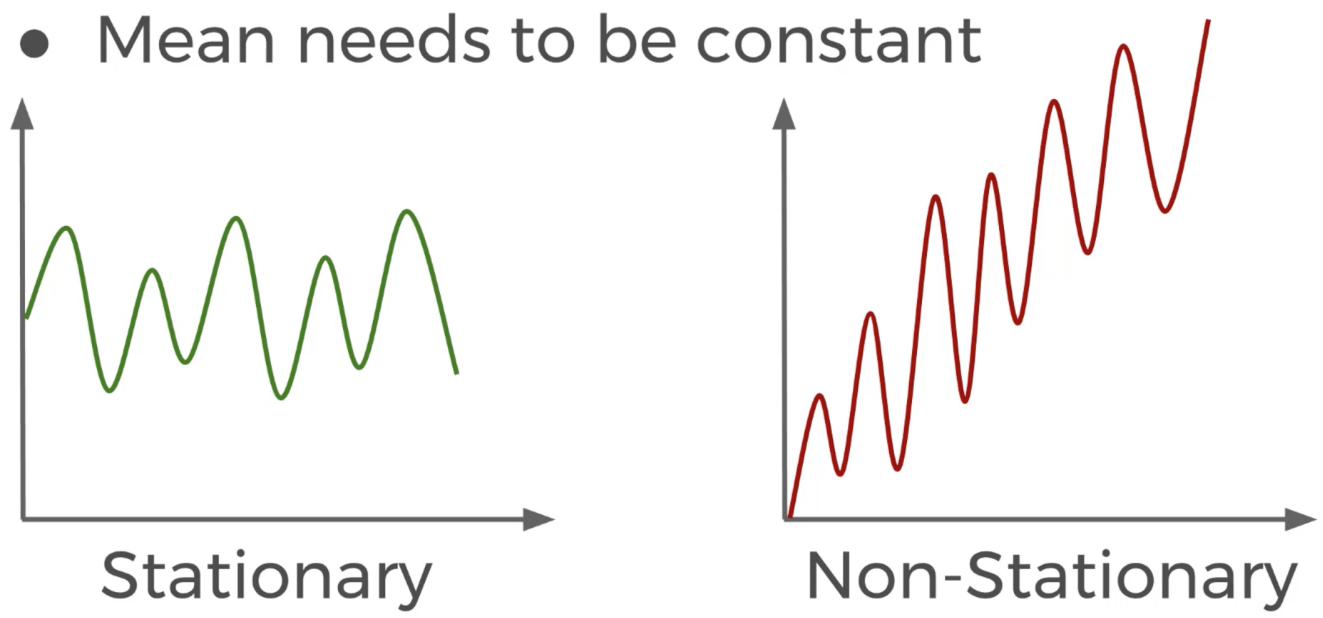
A stationary data set will allow our model to predict that the mean and variance will be the same in future periods.

In [167]:
```
# What a stationary and non-stationary Data look like visually?
```

Aspect 1_Mean?

1- Stationary, constant mean;

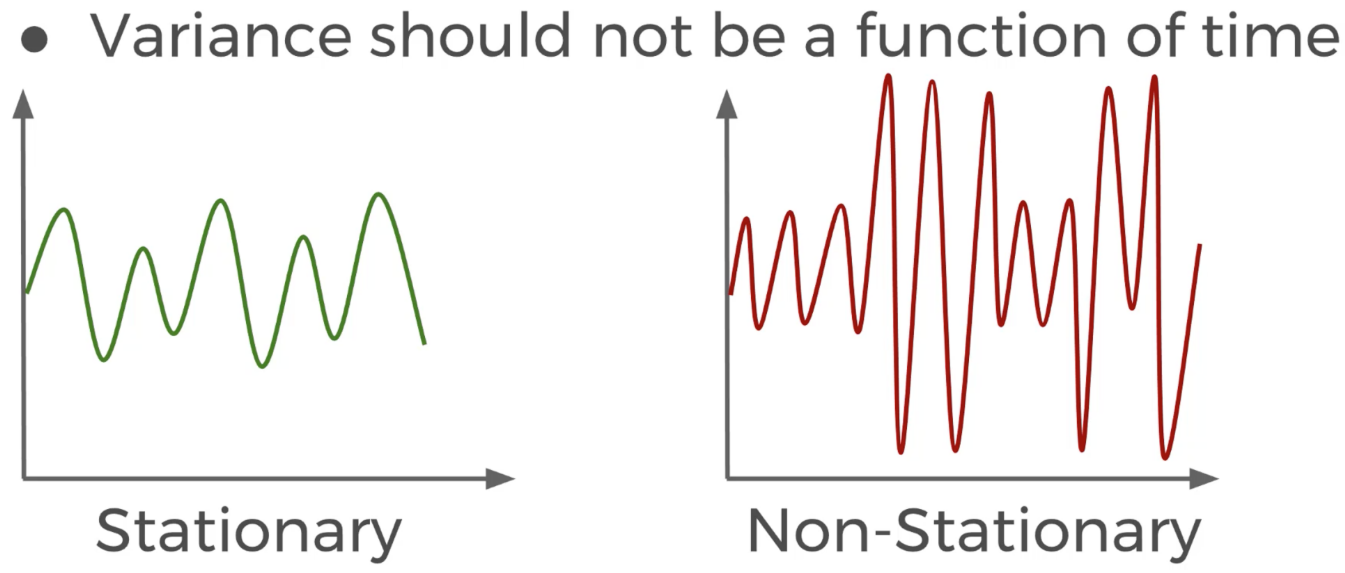
2- Non-stationary, non-constant mean.



Aspect_2: Variance?

1- Stationary Data, variance is not changing over time in function of time

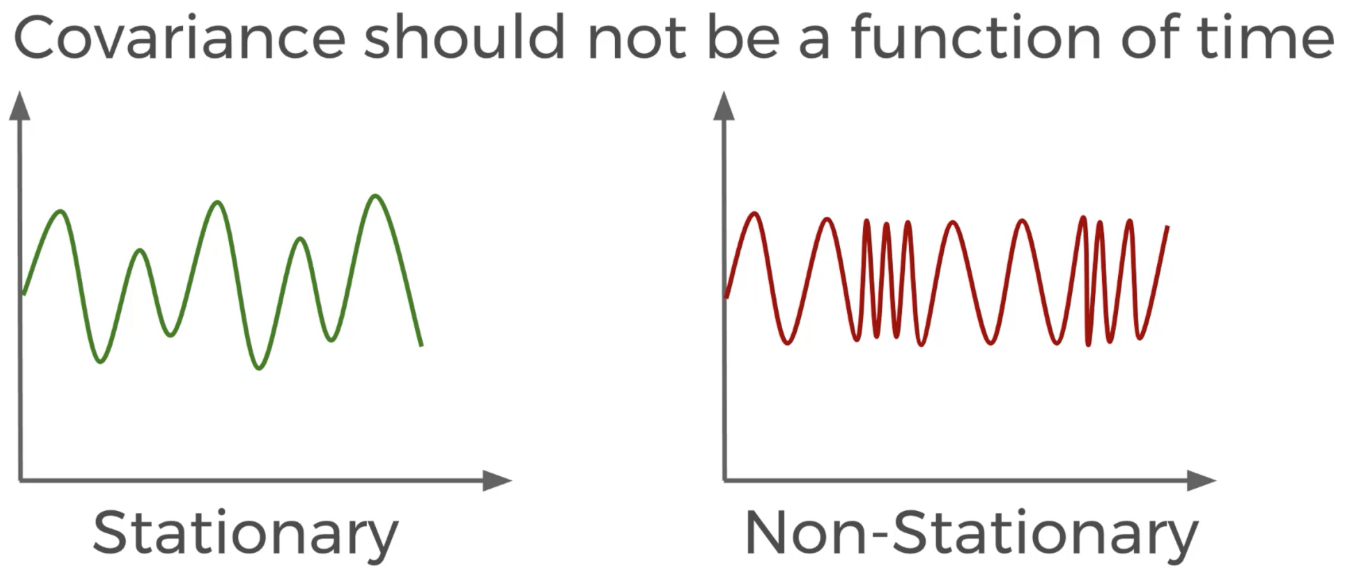
2- Non-stationary Data, variance is changing over time. Variance is a function of time.



Aspect 3_Covariance?

1- Stationary Data, covariance is not changing over time in function of time

2- Non-stationary Data, covariance is not changing over time.




```
In [168]: # What a stationary and non-stationary Data look like mathematically?
```

There are mathematical statsmodel to test stationarity in Data as Augmented Dickey-Fuller test with Pyhton's statsmodels.

```
In [169]: # Conclusion
```

If you've determined your data is not stationary (visually or mathematically), you will then need to transform it to be stationary in order to evaluate it and what type of ARIMA terms you will use. One simple way to do this is hrough "differencing".

In [170]:
```
# General process for ARIMA models
```

General Process for Arima models is the following:

+ Get the Time Series Data
+ Clean the Time Series Data
+ Visulize the Time Series Data
+ Make the time data stationary
+ Plot the Corelation and Autocorrelation Charts
+ Construct the ARIMA Model
+ Use the model to make predictions

In [171]:
```
import numpy as np
import pandas as pd
import statsmodels.api as sm
import statsmodels.formula.api as smf
from statsmodels.tsa.seasonal import seasonal_decompose
```

In [172]:
```
import matplotlib.pyplot as plt
%matplotlib inline
```

In [173]:
```
# Register converters to avoid warnings
pd.plotting.register_matplotlib_converters()
plt.rc("figure", figsize=(16,8))
plt.rc("font", size=14)
```

Step 1: Get and Clean the Data

A-Get the Data

In [174]:
```
# Get and check the Data
df=pd.read_csv('stock_px_2.csv')
```

In [175]:
```
# check the head of DataFrame df
df.head()
```

Out[175]:

	Unnamed: 0	AAPL	MSFT	XOM	SPX
0	2003-01-02 00:00:00	7.40	21.11	29.22	909.03
1	2003-01-03 00:00:00	7.45	21.14	29.24	908.59
2	2003-01-06 00:00:00	7.45	21.52	29.96	929.01
3	2003-01-07 00:00:00	7.43	21.93	28.95	922.93
4	2003-01-08 00:00:00	7.28	21.31	28.83	909.93

In [176]:
```
# check the index of the columns
df.columns
```

Out[176]:
```
Index(['Unnamed: 0', 'AAPL', 'MSFT', 'XOM', 'SPX'], dtype='object')
```

B_Clean The Data

In [177]:
```
# B_1: Get rid of the coulmn name 'unnamed:0'
df1 = pd.read_csv('stock_px_2.csv')
df1.rename(columns={'Unnamed: 0':'Dates'}))
```

Out[177]:

	Dates	AAPL	MSFT	XOM	SPX
0	2003-01-02 00:00:00	7.40	21.11	29.22	909.03
1	2003-01-03 00:00:00	7.45	21.14	29.24	908.59
2	2003-01-06 00:00:00	7.45	21.52	29.96	929.01
3	2003-01-07 00:00:00	7.43	21.93	28.95	922.93
4	2003-01-08 00:00:00	7.28	21.31	28.83	909.93
...	...	...	...	...	...
2209	2011-10-10 00:00:00	388.81	26.94	76.28	1194.89
2210	2011-10-11 00:00:00	400.29	27.00	76.27	1195.54
2211	2011-10-12 00:00:00	402.19	26.96	77.16	1207.25
2212	2011-10-13 00:00:00	408.43	27.18	76.37	1203.66
2213	2011-10-14 00:00:00	422.00	27.27	78.11	1224.58

2214 rows × 5 columns

In [178]:
```
# B_2: Get rid of the coulmn name 'unnamed:0'
df2=df1.rename(columns={'Unnamed: 0':'Dates'}))
```

In [179]:
```
# check the head of DataFrame df2
df2.head()
```

Out[179]:

	Dates	AAPL	MSFT	XOM	SPX
0	2003-01-02 00:00:00	7.40	21.11	29.22	909.03
1	2003-01-03 00:00:00	7.45	21.14	29.24	908.59
2	2003-01-06 00:00:00	7.45	21.52	29.96	929.01
3	2003-01-07 00:00:00	7.43	21.93	28.95	922.93
4	2003-01-08 00:00:00	7.28	21.31	28.83	909.93

In [180]:
```
# check the tail of DataFrame df2
df2.tail()
```

Out[180]:

	Dates	AAPL	MSFT	XOM	SPX
2209	2011-10-10 00:00:00	388.81	26.94	76.28	1194.89
2210	2011-10-11 00:00:00	400.29	27.00	76.27	1195.54
2211	2011-10-12 00:00:00	402.19	26.96	77.16	1207.25
2212	2011-10-13 00:00:00	408.43	27.18	76.37	1203.66
2213	2011-10-14 00:00:00	422.00	27.27	78.11	1224.58

In [181]:
```
# B_3: Convert Dates column to a datetime column
df2['Dates']= pd.to_datetime(df2['Dates'])
```

In [182]:
```
df2.head()
```

Out[182]:

	Dates	AAPL	MSFT	XOM	SPX
0	2003-01-02	7.40	21.11	29.22	909.03
1	2003-01-03	7.45	21.14	29.24	908.59
2	2003-01-06	7.45	21.52	29.96	929.01
3	2003-01-07	7.43	21.93	28.95	922.93
4	2003-01-08	7.28	21.31	28.83	909.93

In [183]:
```
df2.head()
```

Out[183]:

	Dates	AAPL	MSFT	XOM	SPX
0	2003-01-02	7.40	21.11	29.22	909.03
1	2003-01-03	7.45	21.14	29.24	908.59
2	2003-01-06	7.45	21.52	29.96	929.01
3	2003-01-07	7.43	21.93	28.95	922.93
4	2003-01-08	7.28	21.31	28.83	909.93

In [184]:
```
# check the format of the index
df2.index
```

Out[184]:
```
RangeIndex(start=0, stop=2214, step=1)
```

In [185]:
```
# Describe the parameters of df2
df2.describe().transpose()
```

Out[185]:

	count	mean	std	min	25%	50%	75%	max
AAPL	2214.0	125.516147	107.394693	6.56	37.1350	91.455	185.6050	422.00
MSFT	2214.0	23.945452	3.255198	14.33	21.7000	24.000	26.2800	34.07
XOM	2214.0	59.558744	16.725025	26.21	49.4925	62.970	72.5100	87.48
SPX	2214.0	1183.773311	180.983466	676.53	1077.0600	1189.260	1306.0575	1565.15

Step 2: Visualize the Time Series Data

In [186]:
```
# Transform DataFrames to Time series
s1_AAPL = df['AAPL']
s2_MSFT = df['MSFT']
s3_XOM = df['XOM']
s4_SPX = df['SPX']
```

In [187]:
```
# check of the type of time_series
type (s1_AAPL)
type(s2_MSFT)
type(s3_XOM)
type(s4_SPX)
```

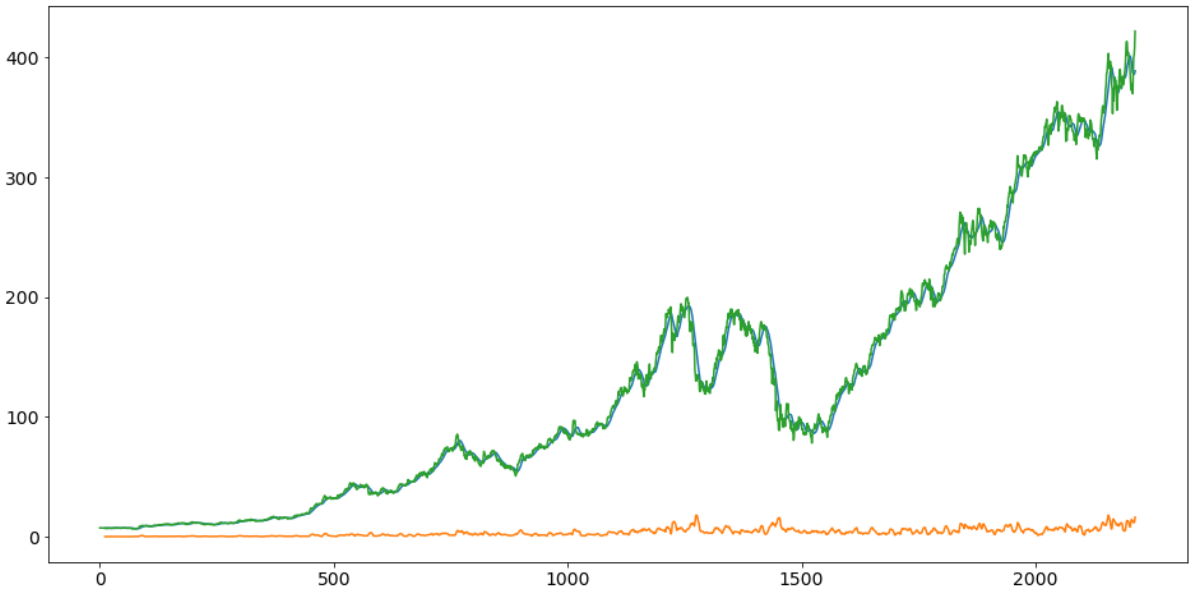
Out[187]:
```
pandas.core.series.Series
```

In [188]:
```
# Display the head of the series
s1_AAPL.head(),s2_MSFT.head(),s3_XOM.head(),s4_SPX.head()
```

Out[188]:
```
(0    7.40
 1    7.45
 2    7.45
 3    7.43
 4    7.28
 Name: AAPL, dtype: float64,
 0    21.11
 1    21.14
 2    21.52
 3    21.93
 4    21.31
 Name: MSFT, dtype: float64,
 0    29.22
 1    29.24
 2    29.96
 3    28.95
 4    28.83
 Name: XOM, dtype: float64,
 0    909.03
 1    908.59
 2    929.01
 3    922.93
 4    909.93
 Name: SPX, dtype: float64)
```

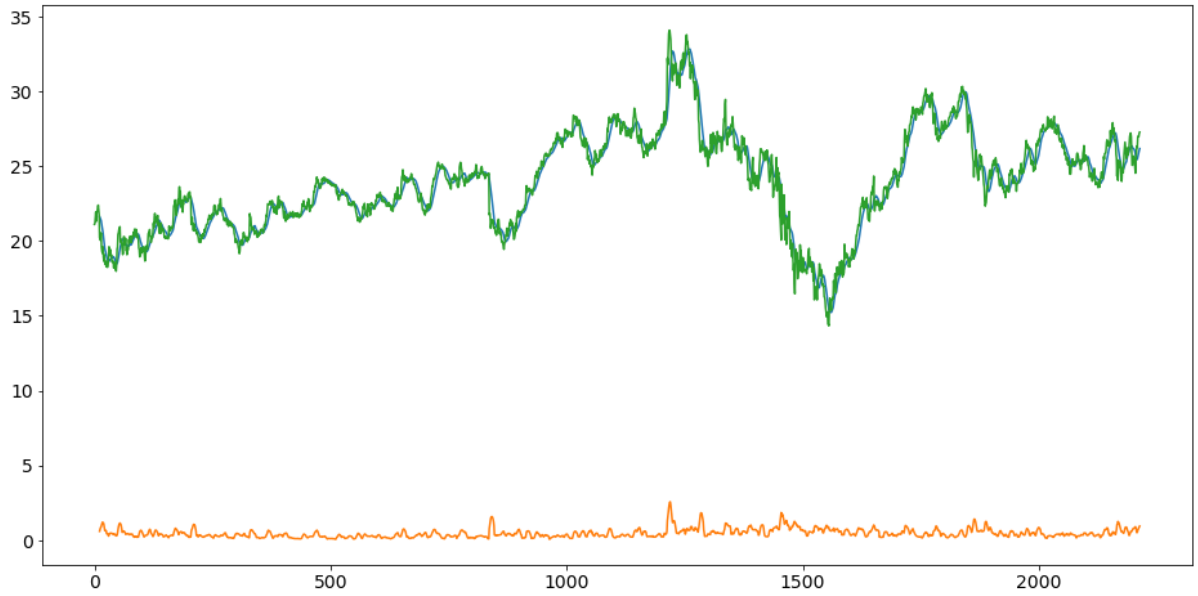
In [189]:
```
s1_AAPL.rolling(12).mean().plot(label='12 Month Rolling Mean')
s1_AAPL.rolling(12).std().plot(label='12 Month Rolling std')
s1_AAPL.plot()
plt.legend
#plot_size_inches(20,50)
```

Out[189]:
```
<function matplotlib.pyplot.legend(*args, **kwargs)>
```



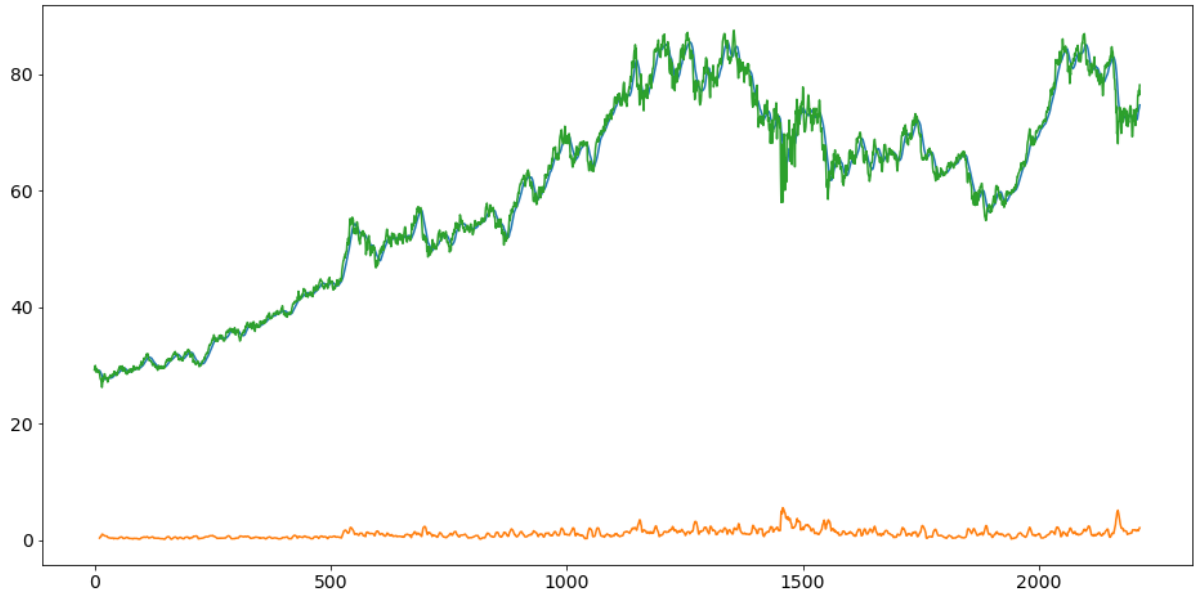
In [190]:
```
s2_MSFT.rolling(12).mean().plot(label='12 Month Rolling Mean')
s2_MSFT.rolling(12).std().plot(label='12 Month Rolling std')
s2_MSFT.plot(), plt.legend
```

Out[190]:
```
(<matplotlib.axes._subplots.AxesSubplot at 0x7ff4a9010880>,
 <function matplotlib.pyplot.legend(*args, **kwargs)>)
```



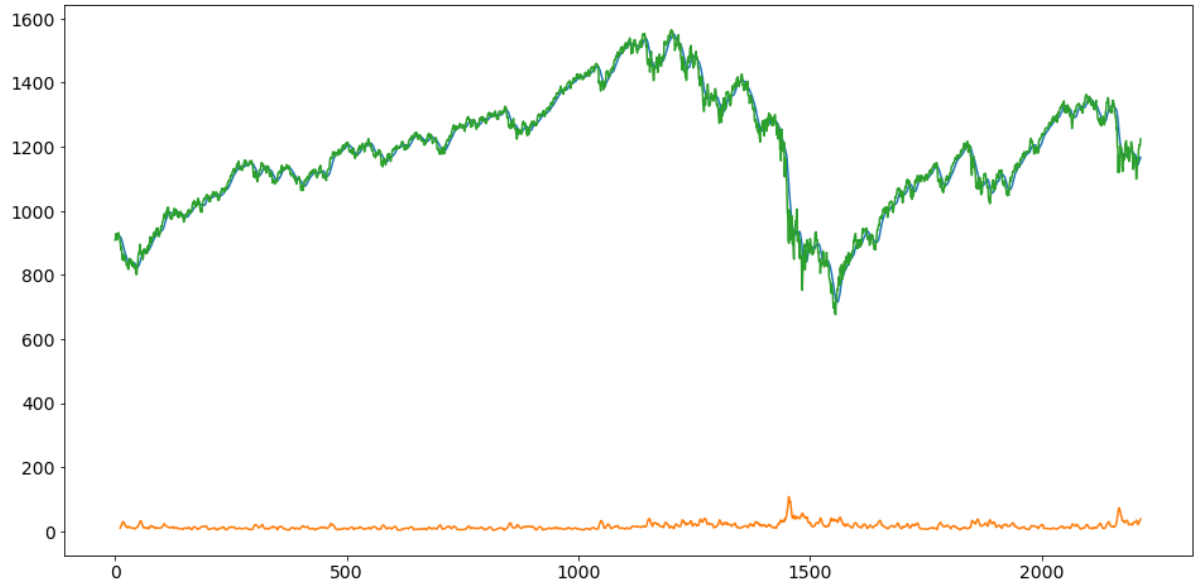
In [191]:
```
s3_XOM.rolling(12).mean().plot(label='12 Month Rolling Mean')
s3_XOM.rolling(12).std().plot(label='12 Month Rolling std')
s3_XOM.plot(), plt.legend
```

Out[191]:
```
(<matplotlib.axes._subplots.AxesSubplot at 0x7ff4abd28580>,
 <function matplotlib.pyplot.legend(*args, **kwargs)>)
```



In [192]:
```
s4_SPX.rolling(12).mean().plot(label='12 Month Rolling Mean')
s4_SPX.rolling(12).std().plot(label='12 Month Rolling std')
s4_SPX.plot()
plt.legend
```

Out[192]:
```
<function matplotlib.pyplot.legend(*args, **kwargs)>
```



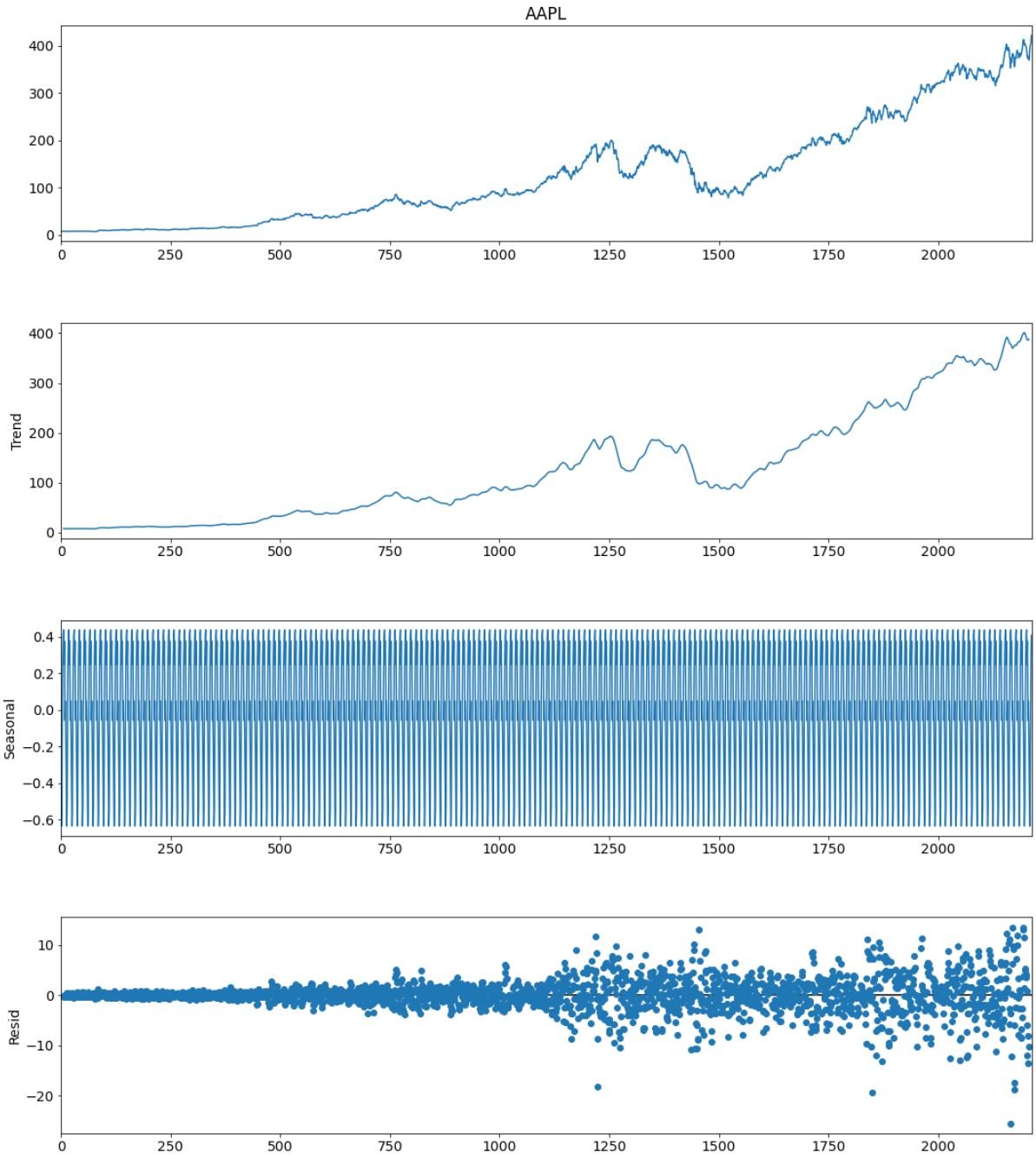
In [193]:
```
s1_AAPL.index, s2_MSFT.index, s3_XOM.index, s4_SPX.index
```

Out[193]:
```
(RangeIndex(start=0, stop=2214, step=1),
 RangeIndex(start=0, stop=2214, step=1),
 RangeIndex(start=0, stop=2214, step=1),
 RangeIndex(start=0, stop=2214, step=1))
```

In [194]:
```
# Error trend and decomposition
from statsmodels.tsa.seasonal import seasonal_decompose
```

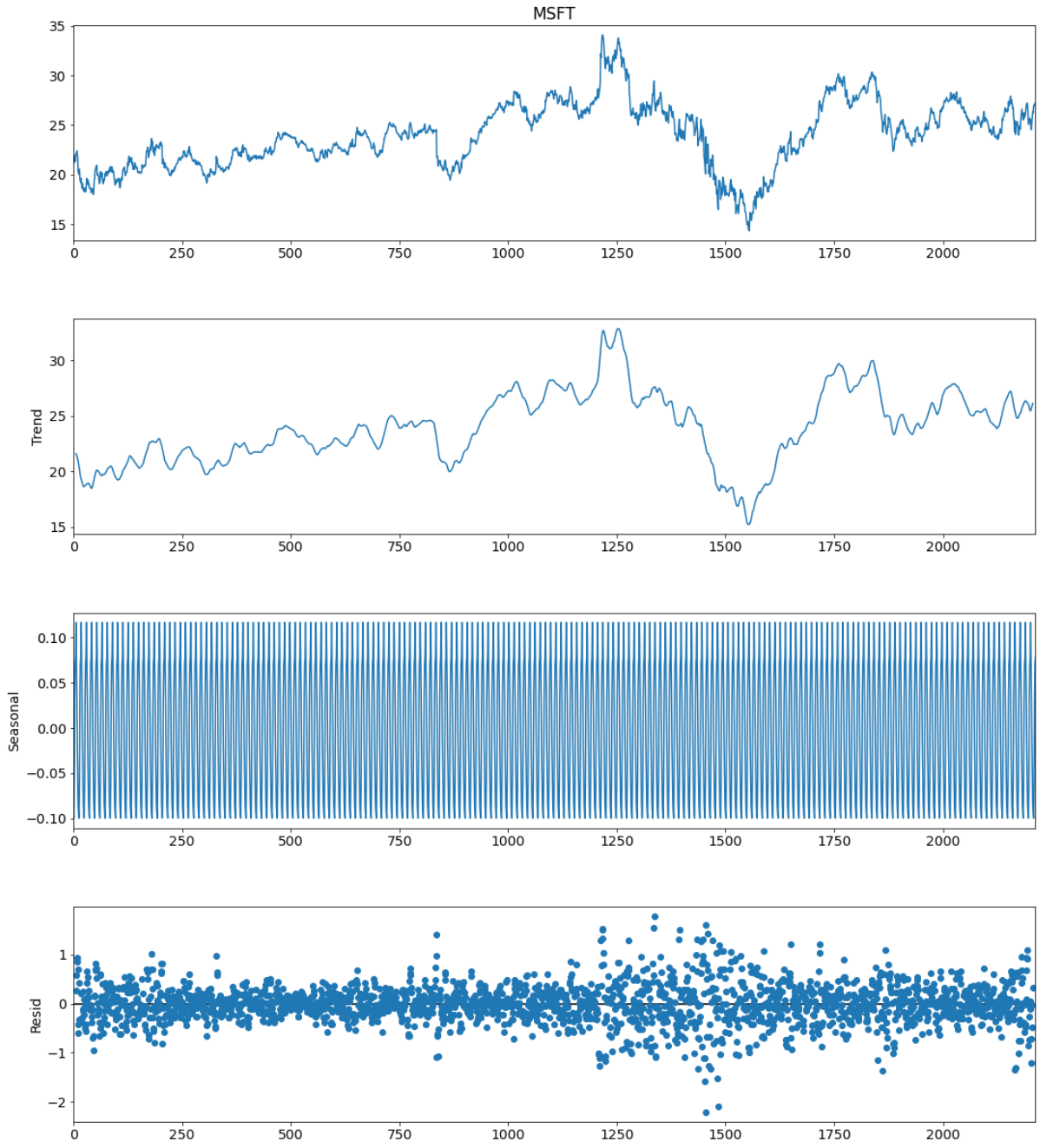
In [195]:
```
decomp_1 = seasonal_decompose(s1_AAPL,freq=12)
Fig_1 = decomp_1.plot()
Fig_1.set_size_inches(15,18)
```

```
<ipython-input-195-e59dac4c3818>:1: FutureWarning:

the 'freq'' keyword is deprecated, use 'period' instead
```



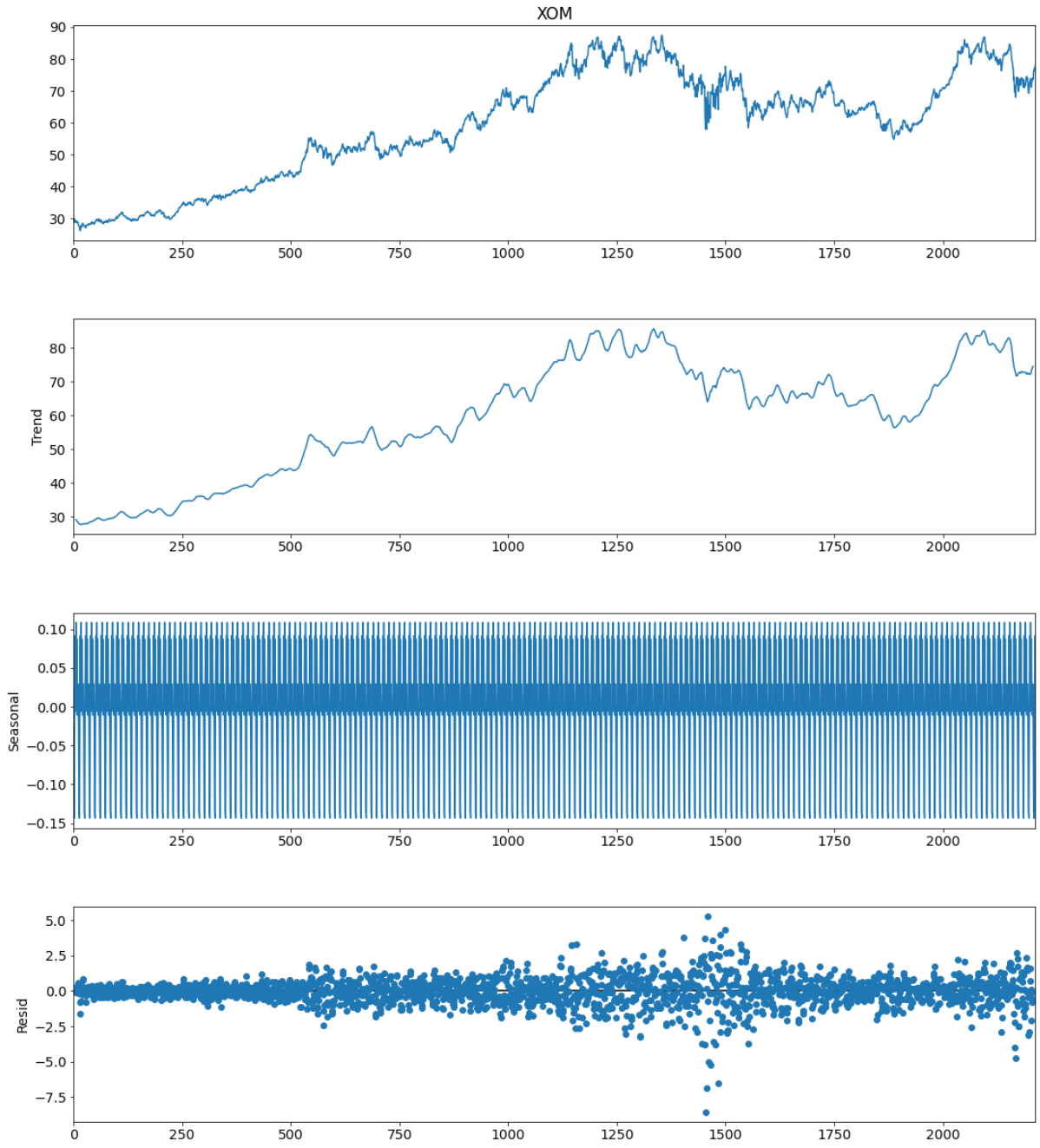
In [196]:
```
decomp_2 = seasonal_decompose(s2_MSFT,freq=12)
Fig_2 = decomp_2.plot()
Fig_2.set_size_inches(15,18)
```

```
<ipython-input-196-5d075d18bd76>:1: FutureWarning:

the 'freq'' keyword is deprecated, use 'period' instead
```



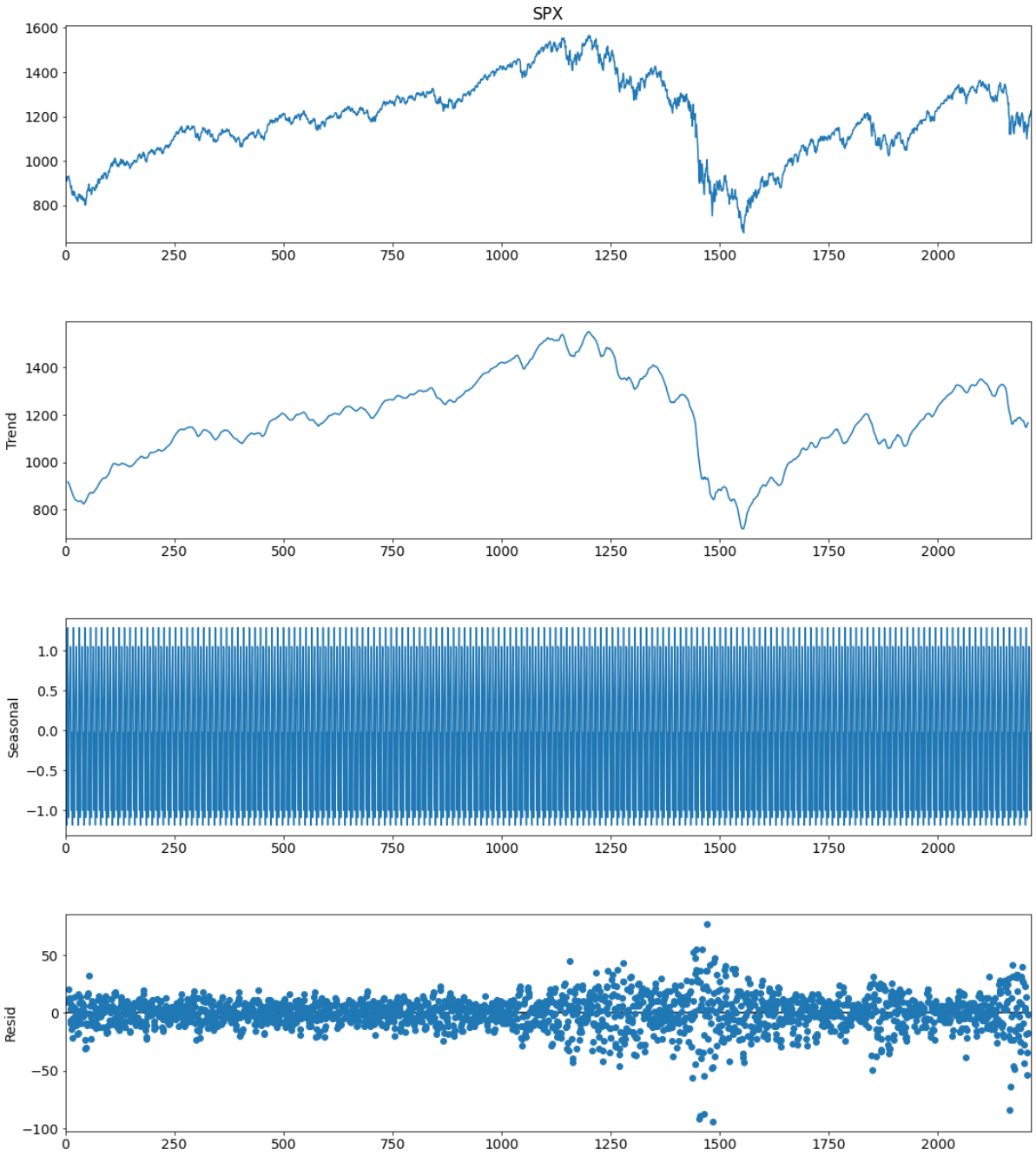
In [197]:
```
decomp_3 = seasonal_decompose(s3_XOM,freq=12)
Fig_3 = decomp_3.plot()
Fig_3.set_size_inches(15,18)
```

```
<ipython-input-197-499ddd5c0bd5>:1: FutureWarning:

the 'freq'' keyword is deprecated, use 'period' instead
```



In [198]:
```
decomp_4 = seasonal_decompose(s4_SPX,freq=13)
Fig_4 = decomp_4.plot()
Fig_4.set_size_inches(15,18)
```

```
<ipython-input-198-a0bdad47166b>:1: FutureWarning:

the 'freq'' keyword is deprecated, use 'period' instead
```



Step 3: Test Data stationarity mathematically with Pyhton's statsmodels

Augmented Dickey-Fuller test

In [199]:
```
# check the DataFrame
df2.head()
```

Out[199]:

	Dates	AAPL	MSFT	XOM	SPX
0	2003-01-02	7.40	21.11	29.22	909.03
1	2003-01-03	7.45	21.14	29.24	908.59
2	2003-01-06	7.45	21.52	29.96	929.01
3	2003-01-07	7.43	21.93	28.95	922.93
4	2003-01-08	7.28	21.31	28.83	909.93

Is the Data stationary or not Stationary?

''' Statistical test used: Augmented Dickey-Fuller unit root test has:

```
    - Null hypothesis: Non stationary time series;
    - Alternative hypothesis: the series has no unit root and actually is stationar
y
We're going to decide on this based off a p value:
    a) returned a small p value typically less than 0.5 indicates strong evidence a
gainst the null hypothesis,
    mean rejection of null hypothesis (Non stationary time series)
    b) returned a large p value, greater than 0.5 that indicates weak evidence agai
nst the null hypothesis,
    mean we fail to reject Null hypothesis'''
```

In [200]:
```
# Runnning Augmented Dickey-Fuller model to test data
from statsmodels.tsa.stattools import adfuller
```

In [201]:
```
result_1_AAPL = adfuller(df2['AAPL'])
```

In [202]:
```
result_1_AAPL
```

Out[202]:
```
(1.5777198065520535,
 0.9977891697734111,
 10,
 2203,
 {'1%': -3.4333218274973816,
  '5%': -2.8628528598160963,
  '10%': -2.5674688992643837},
 11344.462171852701)
```

In [203]:
```
def adf_check(s1_AAPL):
    result_1_AAPL = adfuller(s1_AAPL)
print("Augmented Dicky-Fuller Test")
labels = ['ADF Test Statistic','p-value','# of lags', 'Num of observations use
d', 'AIC','X?']

for value,label in zip(result_1_AAPL,labels):
    print(label+" : "+str(value))

if result_1_AAPL [1] <= 0.05:
        print("Strong evidence against null hypothesis")
        print('reject null hypothesis')
        print("AAPL Data has no unit root and is stationary")

else:
        print('weak evidence against null hypothesis')
        print ('fail to reject null hypothesis')
        print ('AAPL Data has a unit root, it is non-stationary')
```

```
Augmented Dicky-Fuller Test
ADF Test Statistic : 1.5777198065520535
p-value : 0.9977891697734111
# of lags : 10
Num of observations used : 2203
AIC : {'1%': -3.4333218274973816, '5%': -2.8628528598160963, '10%': -2.567468
8992643837}
X? : 11344.462171852701
weak evidence against null hypothesis
fail to reject null hypothesis
AAPL Data has a unit root, it is non-stationary
```

In [204]:
```
result_2_MSFT = adfuller(df2['MSFT'])
```

In [205]:
```
result_2_MSFT
```

Out[205]:
```
(-2.8730860675303287,
 0.04856843065105696,
 23,
 2190,
 {'1%': -3.43333948922474,
  '5%': -2.8628606583021843,
  '10%': -2.5674730514376263},
 2198.9148942917464)
```

In [206]:
```
def adf_check(s2_MSFT):
    result_2_MSFT = adfuller(s2_MSFT)
print("Augmented Dicky-Fuller Test")
labels = ['ADF Test Statistic','p-value','# of lags', 'Num of observations use
d', 'AIC','X?']

for value,label in zip(result_2_MSFT ,labels):
    print(label+" : "+str(value))

if result_2_MSFT  [1] <= 0.05:
        print("Strong evidence against null hypothesis")
        print('reject null hypothesis')
        print("MSFT Data has no unit root and is stationary")

else:
        print('weak evidence against null hypothesis')
        print ('fail to reject null hypothesis')
        print ('MSFT Data has a unit root, it is non-stationary')
```

```
Augmented Dicky-Fuller Test
ADF Test Statistic : -2.8730860675303287
p-value : 0.04856843065105696
# of lags : 23
Num of observations used : 2190
AIC : {'1%': -3.43333948922474, '5%': -2.8628606583021843, '10%': -2.56747305
14376263}
X? : 2198.9148942917464
Strong evidence against null hypothesis
reject null hypothesis
MSFT Data has no unit root and is stationary
```

In [207]:
```
result_s3_XOM = adfuller(df2['XOM'])
```

In [208]:
```
result_s3_XOM
```

Out[208]:
```
(-1.650451458274292,
 0.45680498648995127,
 18,
 2195,
 {'1%': -3.4333326714656223,
  '5%': -2.8628576479452597,
  '10%': -2.5674714486226202},
 6480.729901763714)
```

In [209]:
```
def adf_check(s3_XOM):
    result_s3_XOM = adfuller(s3_XOM)
print("Augmented Dicky-Fuller Test")
labels = ['ADF Test Statistic','p-value','# of lags', 'Num of observations use
d', 'AIC','X?']

for value,label in zip(result_s3_XOM ,labels):
    print(label+" : "+str(value))

if  result_s3_XOM  [1] <= 0.05:
        print("Strong evidence against null hypothesis")
        print('reject null hypothesis')
        print("XOM Data has no unit root and is stationary")

else:
        print('weak evidence against null hypothesis')
        print ('fail to reject null hypothesis')
        print ('XOM Data has a unit root, it is non-stationary')
```

```
Augmented Dicky-Fuller Test
ADF Test Statistic : -1.650451458274292
p-value : 0.45680498648995127
# of lags : 18
Num of observations used : 2195
AIC : {'1%': -3.4333326714656223, '5%': -2.8628576479452597, '10%': -2.567471
4486226202}
X? : 6480.729901763714
weak evidence against null hypothesis
fail to reject null hypothesis
XOM Data has a unit root, it is non-stationary
```

In [210]:
```
result_s4_SPX = adfuller(df2['SPX'])
```

In [211]:
```
result_s4_SPX
```

Out[211]:
```
(-2.1827537567644626,
 0.21256428346615042,
 22,
 2191,
 {'1%': -3.433338123180619,
  '5%': -2.862860055130884,
  '10%': -2.567472730288902},
 17849.293768680058)
```

In [212]:
```
def adf_check(s4_SPX):
    result_s4_SPX = adfuller(s4_SPX)
print("Augmented Dicky-Fuller Test")
labels = ['ADF Test Statistic','p-value','# of lags', 'Num of observations use
d', 'AIC','X?']

for value,label in zip(result_s4_SPX ,labels):
    print(label+" : "+str(value))

if result_s4_SPX  [1] <= 0.05:
        print("Strong evidence against null hypothesis")
        print('reject null hypothesis')
        print("SPX Data has no unit root and is stationary")

else:
        print('weak evidence against null hypothesis')
        print ('fail to reject null hypothesis')
        print ('SPX Data has a unit root, it is non-stationary')
```

```
Augmented Dicky-Fuller Test
ADF Test Statistic : -2.1827537567644626
p-value : 0.21256428346615042
# of lags : 22
Num of observations used : 2191
AIC : {'1%': -3.433338123180619, '5%': -2.862860055130884, '10%': -2.56747273
0288902}
X? : 17849.293768680058
weak evidence against null hypothesis
fail to reject null hypothesis
SPX Data has a unit root, it is non-stationary
```

Augmented Dicky-Fuller Test

Stock	AAPL NON-STATIONARY	MSFT STATIONARY	XOM NON-STATIONARY	SPX NON-STATIONARY
ADF Test Statistic	1.5777198065520535	-2.8730860675303287	-1.650451458274292	-2.1827537567644626
p-value	0.9977891697734111	0.04856843065105696	0.45680498648995127	0.21256428346615042
# of lags	10	23	18	22
Num of observations used	2203	2190	2195	2191
AIC	{'1%': -3.4333218274973816, '5%': -2.8628528598160963, '10%': -2.5674688992643837}	{'1%': -3.43333948922474, '5%': -2.8628606583021843, '10%': -2.5674730514376263}	{'1%': -3.4333326714656223, '5%': -2.8628576479452597, '10%': -2.567471448622620 2}	{'1%': -3.433338123180619, '5%': -2.862860055130884, '10%': -2.567472730288902}
X	11344.462171852701	2198.9148942917464	6480.729901763714	17849.293768680058
Stationary or non-Stationary	• Weak evidence against null hypothesis • Fail to reject null hypothesis **AAPL Data has a unit root, it is non-stationary**	• Strong evidence against null hypothesis • reject null hypothesis **MSFT Data has no unit root, and it is stationary**	• weak evidence against null hypothesis • fail to reject null hypothesis **XOM Data has a unit root, and it is non-stationary**	• weak evidence against null hypothesis • fail to reject null hypothesis **SPX Data has a unit root, and it is non-stationary**

Note **We have now realized that our data is seasonal (it is also pretty obvious from the plot itself). This means we need to use Seasonal ARIMA on our model. If our data was not seasonal, it means we could use just ARIMA on it. We will take this into account when differencing our data! Typically financial stock data won't be seasonal, but that is kind of the point of this section, to show you common methods, that won't work well on stock finance data!**

Step 4: Make the Time Series Data stationary

DIFFERENCING

1- Non-seasonal ARIMA models are generally denoteed ARIMA (pdq)

where parameteter p,q and q are non-negative integers;

a) AR(p): Autoregression: A basic regression model that utilizes the dependent relationship between a current observation and observations over a previous period;

b) I(d): Integrated: Differencing observations (subtracting and observation at the previous time step) in order to make the time series stationary;

c) MA(q): Moving Average: A model that uses the dependency between an observation and a residual error from a moving average model applied to lagged observations.

I(d): Integrated: Differencing observations (subtracting and observation at the previous time step) in order to make the time series stationary

s1_AAPL; s3_XOM; s4_SPX

In [213]:
```
def adf_check(s1_AAPL):
    result_1_AAPL = adfuller(s1_AAPL)
print("Augmented Dicky-Fuller Test")
labels = ['ADF Test Statistic','p-value','# of lags', 'Num of observations use
d', 'AIC','X?']

for value,label in zip(result_1_AAPL,labels):
    print(label+" : "+str(value))

if result_1_AAPL [1] <= 0.05:
        print("Strong evidence against null hypothesis")
        print('reject null hypothesis')
        print("AAPL Data has no unit root and is stationary")

else:
        print('weak evidence against null hypothesis')
        print ('fail to reject null hypothesis')
        print ('AAPL Data has a unit root, it is non-stationary')
```

```
Augmented Dicky-Fuller Test
ADF Test Statistic : 1.5777198065520535
p-value : 0.9977891697734111
# of lags : 10
Num of observations used : 2203
AIC : {'1%': -3.4333218274973816, '5%': -2.8628528598160963, '10%': -2.567468
8992643837}
X? : 11344.462171852701
weak evidence against null hypothesis
fail to reject null hypothesis
AAPL Data has a unit root, it is non-stationary
```

In [214]:
```
df2
```

Out[214]:

	Dates	AAPL	MSFT	XOM	SPX
0	2003-01-02	7.40	21.11	29.22	909.03
1	2003-01-03	7.45	21.14	29.24	908.59
2	2003-01-06	7.45	21.52	29.96	929.01
3	2003-01-07	7.43	21.93	28.95	922.93
4	2003-01-08	7.28	21.31	28.83	909.93
...	...	...	...	...	...
2209	2011-10-10	388.81	26.94	76.28	1194.89
2210	2011-10-11	400.29	27.00	76.27	1195.54
2211	2011-10-12	402.19	26.96	77.16	1207.25
2212	2011-10-13	408.43	27.18	76.37	1203.66
2213	2011-10-14	422.00	27.27	78.11	1224.58

2214 rows × 5 columns

In [215]:
```
# First difference
# df['Milk First Difference'] = df['Milk in pounds per cow'] - df['Milk in pou
nds per cow'].shift(1)
df2['AAPL First Difference']= (df2['AAPL']-df2['AAPL'].shift(1)).dropna()
```

In [216]:
```
df2
```

Out[216]:

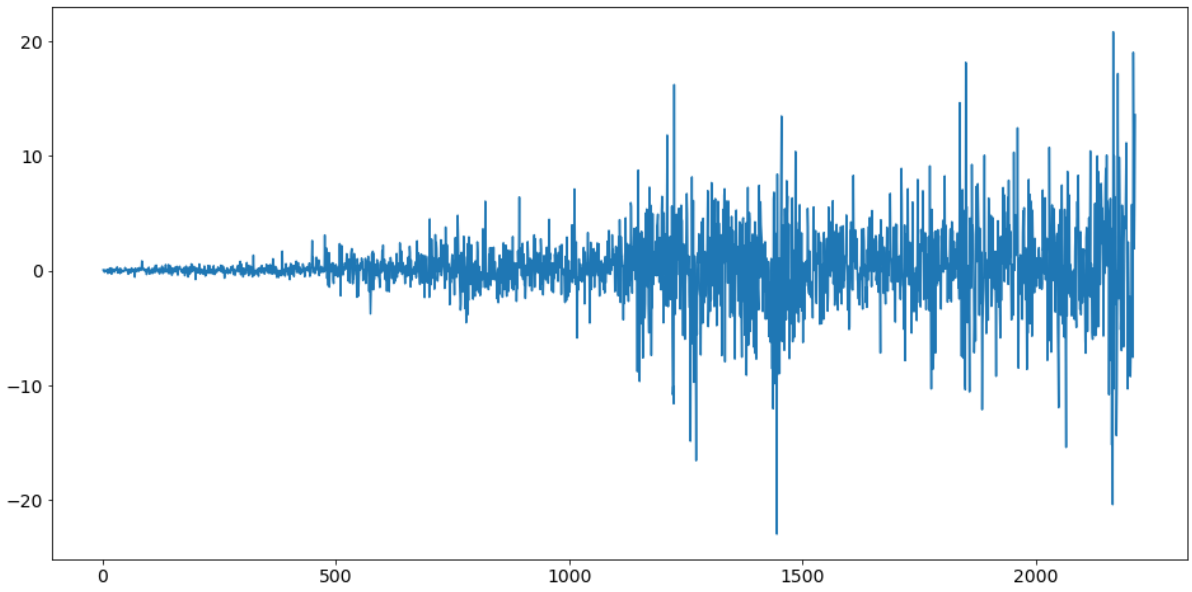
	Dates	AAPL	MSFT	XOM	SPX	AAPL First Difference
0	2003-01-02	7.40	21.11	29.22	909.03	NaN
1	2003-01-03	7.45	21.14	29.24	908.59	0.05
2	2003-01-06	7.45	21.52	29.96	929.01	0.00
3	2003-01-07	7.43	21.93	28.95	922.93	-0.02
4	2003-01-08	7.28	21.31	28.83	909.93	-0.15
...	...	...	...	...	...	...
2209	2011-10-10	388.81	26.94	76.28	1194.89	19.01
2210	2011-10-11	400.29	27.00	76.27	1195.54	11.48
2211	2011-10-12	402.19	26.96	77.16	1207.25	1.90
2212	2011-10-13	408.43	27.18	76.37	1203.66	6.24
2213	2011-10-14	422.00	27.27	78.11	1224.58	13.57

2214 rows × 6 columns

In [217]:
```
#df2[['AAPL','forecast']].plot(figsize=(12,8))
```

In [218]:
```
df2['AAPL First Difference'].plot()
```

Out[218]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x7ff4e094d400>
```



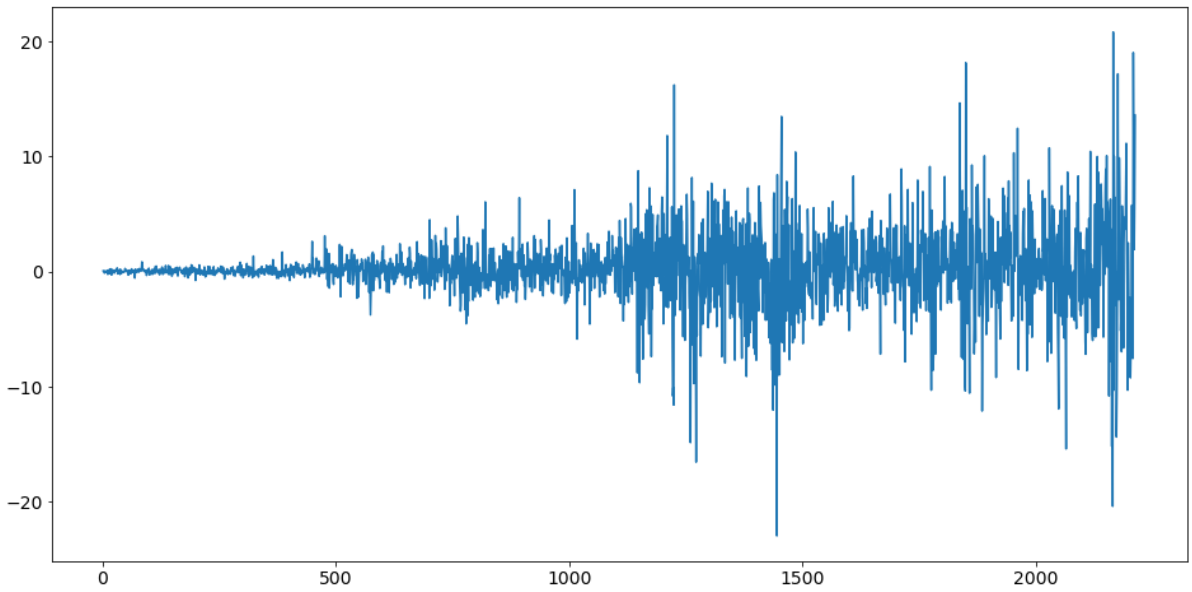
In [219]:
```
# Store in a function for later use!
def adf_check(time_series):
    """
    Pass in a time series, returns ADF report
    """
    result = adfuller(time_series)
    print('Augmented Dickey-Fuller Test:')
    labels = ['ADF Test Statistic','p-value','#Lags Used','Number of Observati
ons Used']

    for value,label in zip(result,labels):
        print(label+' : '+str(value) )

    if result[1] <= 0.05:
        print("strong evidence against the null hypothesis, reject the null hy
pothesis. Data has no unit root and is stationary")
    else:
        print("weak evidence against null hypothesis, time series has a unit r
oot, indicating it is non-stationary ")
```

In [220]:
```
adf_check(df2['AAPL First Difference'].dropna())
```

```
Augmented Dickey-Fuller Test:
ADF Test Statistic : -14.913806700540523
p-value : 1.445218931309417e-27
#Lags Used : 9
Number of Observations Used : 2203
strong evidence against the null hypothesis, reject the null hypothesis. Data
has no unit root and is stationary
```

In [221]:
```
df2['AAPL First Difference'].plot()
```

Out[221]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x7ff4e09243a0>
```



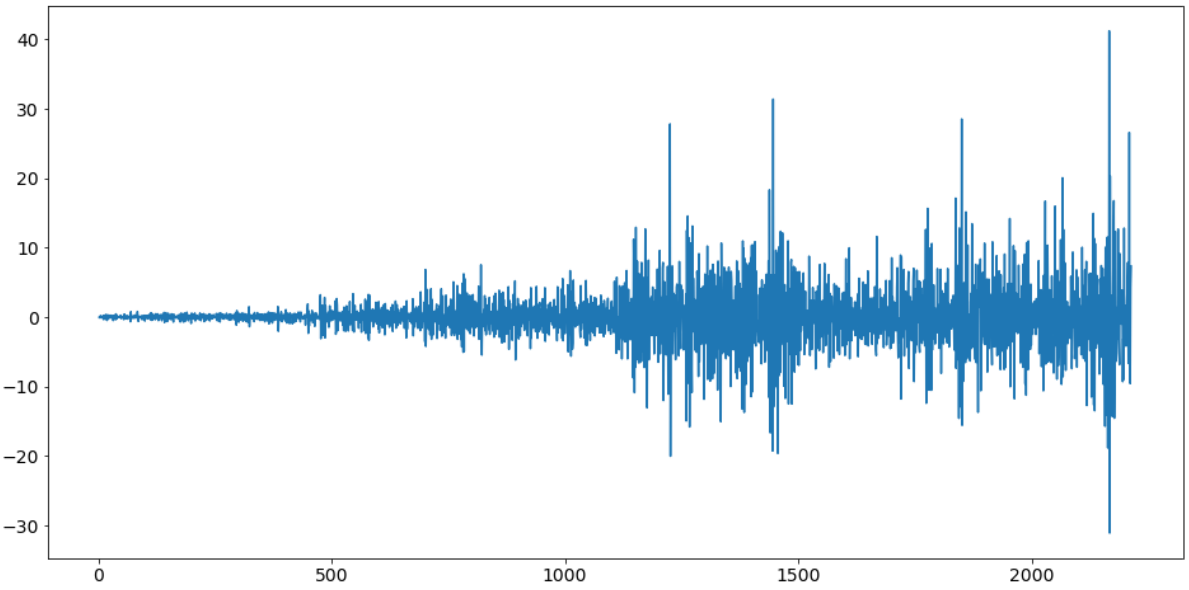
In [222]:
```
# second difference
# Sometimes it would be necessary to do a second difference
# This is just for practice, we didn't need to do a second difference for AAPL
df2['AAPL Second Difference'] = df2['AAPL First Difference'] - df2['AAPL First Difference'].shift(1)
```

In [223]:
```
adf_check(df2['AAPL Second Difference'].dropna())
```

```
Augmented Dickey-Fuller Test:
ADF Test Statistic : -15.499833209223812
p-value : 2.4237307860348852e-28
#Lags Used : 26
Number of Observations Used : 2185
strong evidence against the null hypothesis, reject the null hypothesis. Data has no unit root and is stationary
```

In [224]:
```
df2['AAPL Second Difference'].plot()
```

Out[224]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x7ff4abcd0a30>
```

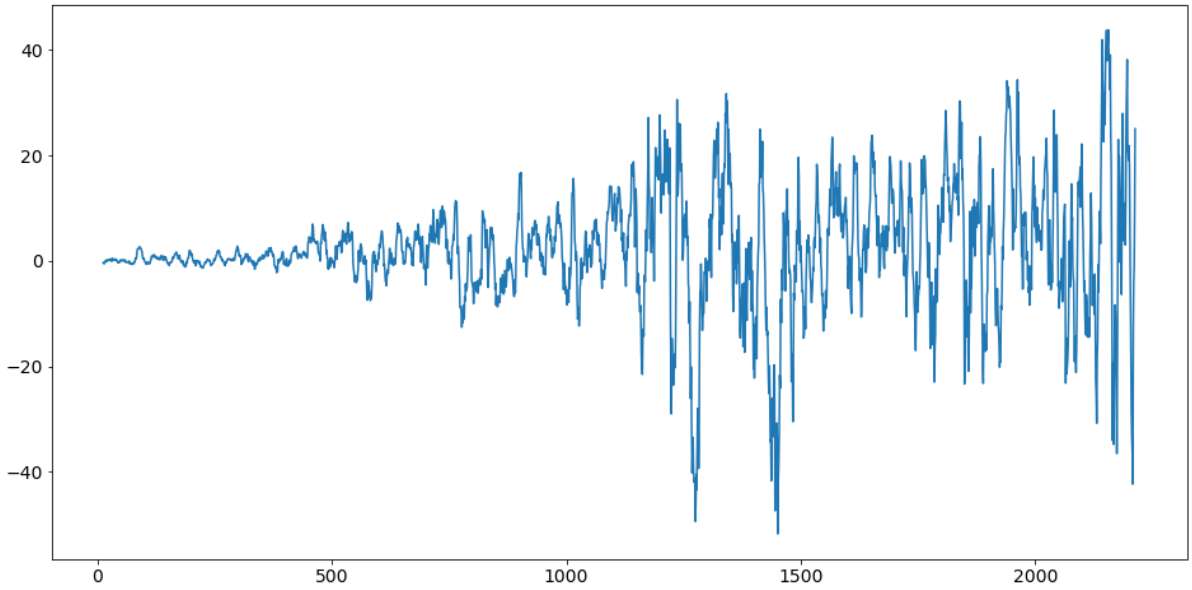


Third Difference: Seasonal Difference

In [225]:
```
df2['Seasonal Difference'] = df2['AAPL']-df2['AAPL'].shift(12)
```

In [226]:
```
df2['Seasonal Difference'].plot()
```

Out[226]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x7ff4ae706be0>
```



In [227]:
```
# Visual Seasonal Difference by itself was not enough!
adf_check(df2['Seasonal Difference'].dropna())
```

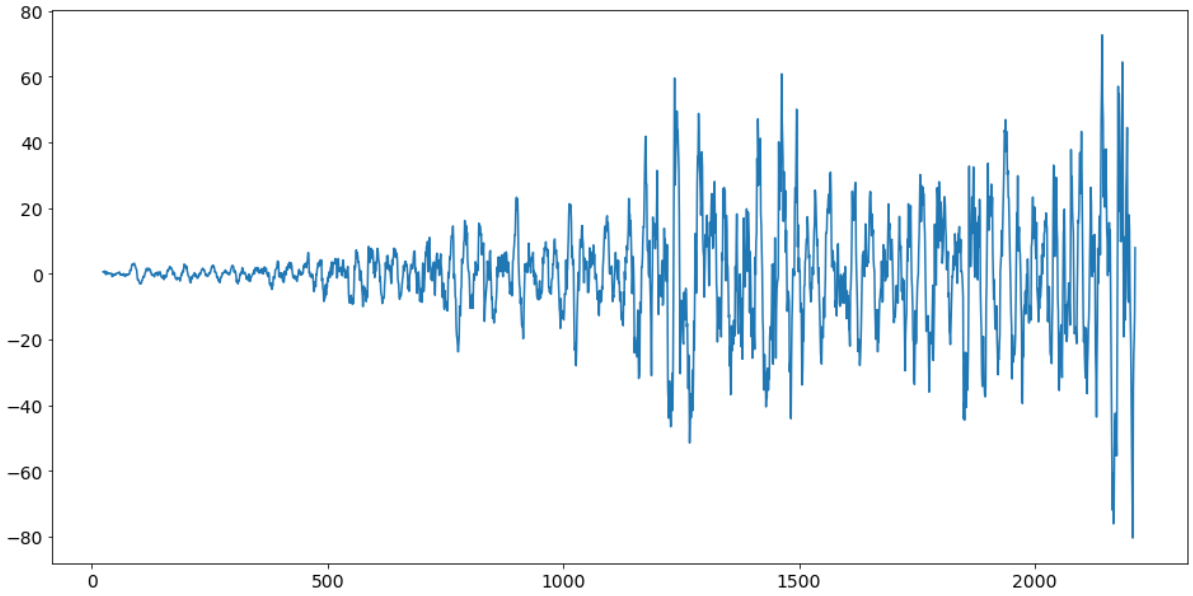
```
Augmented Dickey-Fuller Test:
ADF Test Statistic : -6.598707585682378
p-value : 6.821405535903624e-09
#Lags Used : 26
Number of Observations Used : 2175
strong evidence against the null hypothesis, reject the null hypothesis. Data
has no unit root and is stationary
```

Seasonal First Difference

This is just for practice, we didn't need to do a second difference for AAPL: Seasonal Difference show already the stationarity of data.

In [228]:
```
# You can also do seasonal first difference
df2['Seasonal First Difference'] = df2['Seasonal Difference'] - df2['Seasonal
 Difference'].shift(12)
df2['Seasonal First Difference'].plot()
```

Out[228]: <matplotlib.axes._subplots.AxesSubplot at 0x7ff4e0895c70>



In [229]:
```
adf_check(df2['Seasonal First Difference'].dropna())
```

```
Augmented Dickey-Fuller Test:
ADF Test Statistic : -9.92132912698296
p-value : 2.9855265235847294e-17
#Lags Used : 26
Number of Observations Used : 2163
strong evidence against the null hypothesis, reject the null hypothesis. Data
has no unit root and is stationary
```

Step 5: Plot the Correlation and AutoCorrelation Charts

Autocorrelation and Partial Autocorrelation Plots

An autocorrelation plot (also known as a Correlogram) shows the correlation of the series with itself, lagged by x time units. So the y axis is the correlation and the x axis is the number of time units of lag.

So imagine taking your time series of length T, copying it, and deleting the first observation of copy #1 and the last observation of copy #2. Now you have two series of length T−1 for which you calculate a correlation coefficient. This is the value of of the vertical axis at x=1x=1 in your plots. It represents the correlation of the series lagged by one time unit. You go on and do this for all possible time lags x and this defines the plot.

You will run these plots on your differenced/stationary data. There is a lot of great information for identifying and interpreting ACF and PACF here and here.

Autocorrelation Interpretation The actual interpretation and how it relates to ARIMA models can get a bit complicated, but there are some basic common methods we can use for the ARIMA model. Our main priority here is to try to figure out whether we will use the AR or MA components for the ARIMA model (or both!) as well as how many lags we should use. In general you would use either AR or MA, using both is less common.

If the autocorrelation plot shows positive autocorrelation at the first lag (lag-1), then it suggests to use the AR terms in relation to the lag

If the autocorrelation plot shows negative autocorrelation at the first lag, then it suggests using MA terms.

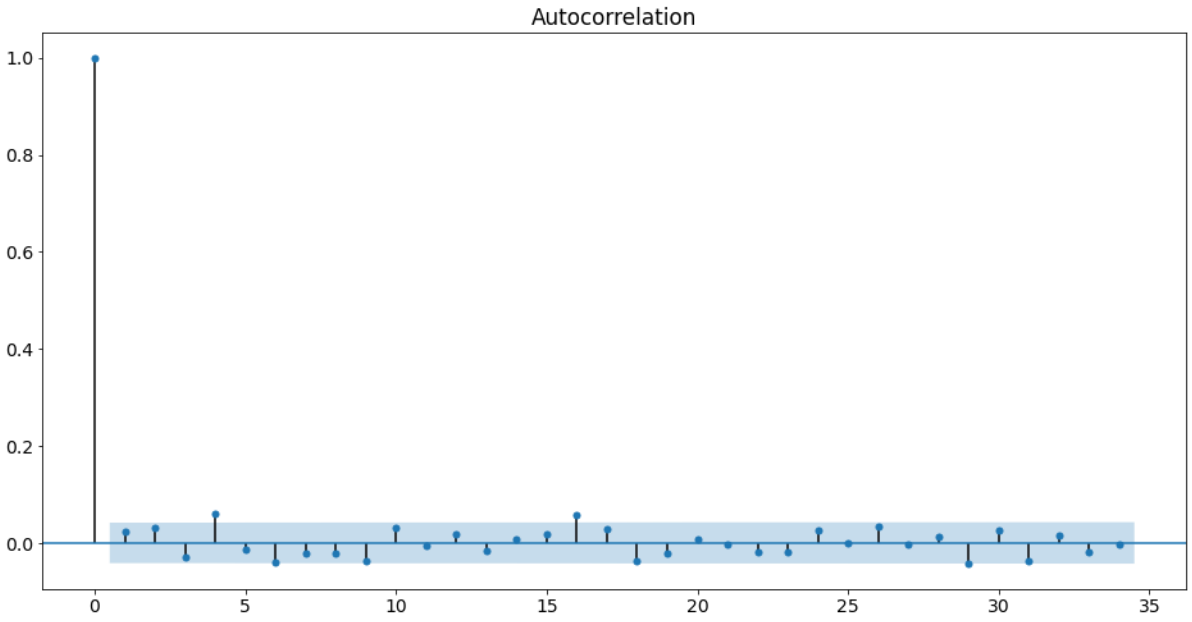
In [230]:
```
from statsmodels.graphics.tsaplots import plot_acf,plot_pacf
```

Note:

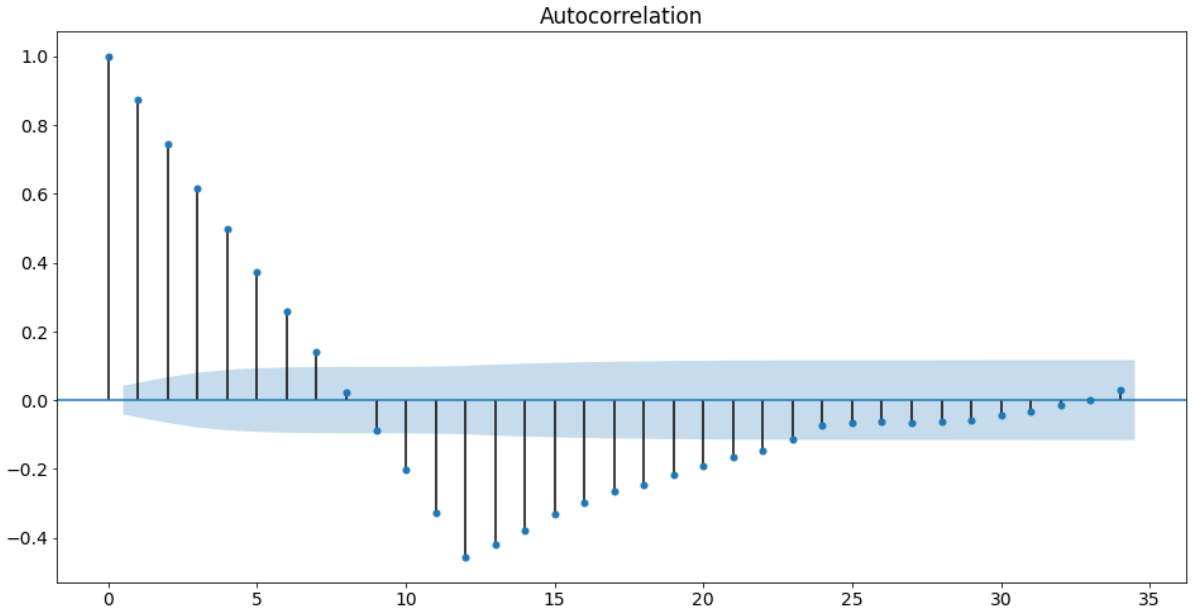
Here we will be showing running the ACF and PACF on multiple differenced data sets that have been made stationary in different ways, typically you would just choose a single stationary data set and continue all the way through with that.

The reason we use two here is to show you the two typical types of behaviour you would see when using ACF.

In [231]:
```
# Duplicate plots
# Check out: https://stackoverflow.com/questions/21788593/statsmodels-duplicate-charts
# https://github.com/statsmodels/statsmodels/issues/1265
fig_first = plot_acf(df2['AAPL First Difference'].dropna())
```



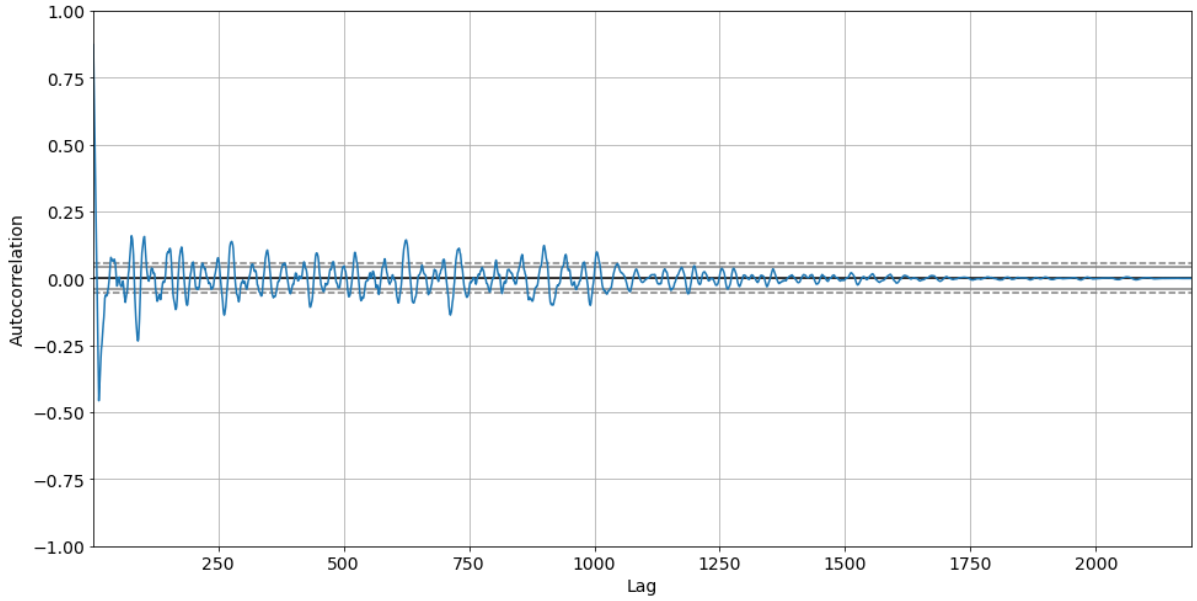
In [232]:
```
fig_seasonal_first = plot_acf(df2["Seasonal First Difference"].dropna())
```



Pandas also has this functionality built in, but only for ACF, not PACF. So I recommend using statsmodels, as ACF and PACF is more core to its functionality than it is to pandas' functionality.

In [233]:
```
from pandas.plotting import autocorrelation_plot
autocorrelation_plot(df2['Seasonal First Difference'].dropna())
```

Out[233]: <matplotlib.axes._subplots.AxesSubplot at 0x7ff4af067760>



Partial Autocorrelation

In general, a partial correlation is a conditional correlation.

It is the correlation between two variables under the assumption that we know and take into account the values of some other set of variables.

For instance, consider a regression context in which y = response variable and x1, x2, and x3 are predictor variables. The partial correlation between y and x3 is the correlation between the variables determined taking into account how both y and x3 are related to x1 and x2.

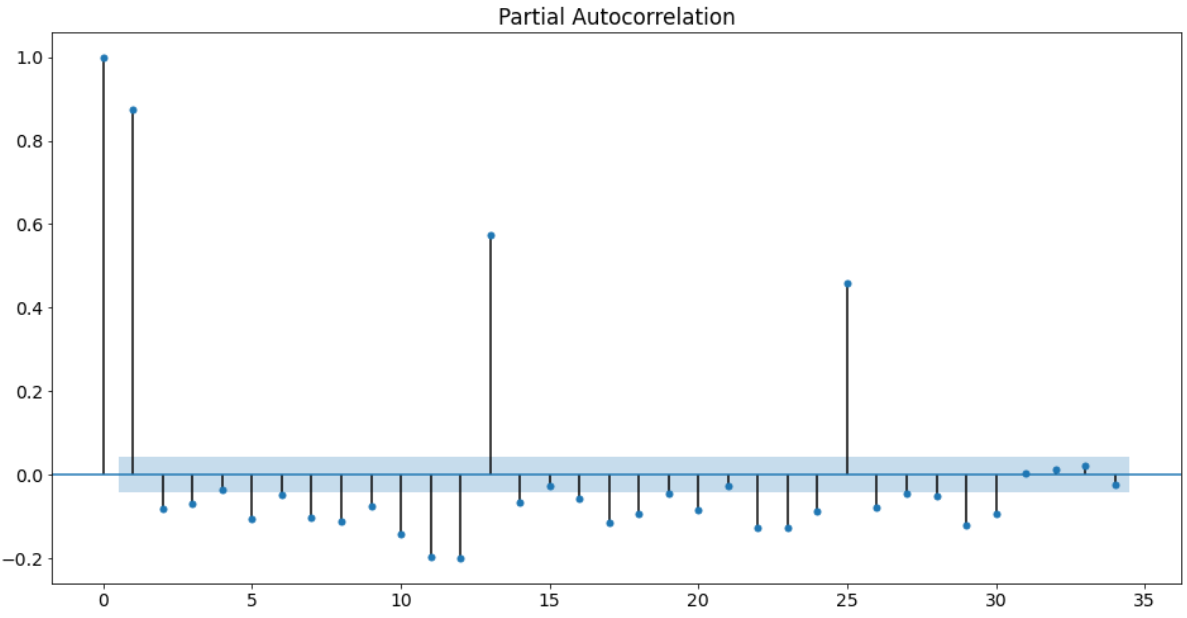
Formally, this is relationship is defined as:

Covariance(y,$x3|x1$,$x2$)Variance($y|x1$,$x2$)Variance($x3|x1$,$x2$)√

For more information about Autocorrellation:

https://www.itl.nist.gov/div898/handbook/pmc/section4/pmc4463.htm
(https://www.itl.nist.gov/div898/handbook/pmc/section4/pmc4463.htm)

In [234]:
```
# We can then plot this relationship/ Partial correlation: pacf
result = plot_pacf(df2["Seasonal First Difference"].dropna())
```



Interpretation

Typically a sharp drop after lag "k" suggests an AR-k model should be used. If there is a gradual decline, it suggests an MA model.

Final Thoughts on Autocorrelation and Partial Autocorrelation

Identification of an AR model is often best done with the PACF. For an AR model, the theoretical PACF “shuts off” past the order of the model. The phrase “shuts off” means that in theory the partial autocorrelations are equal to 0 beyond that point. Put another way, the number of non-zero partial autocorrelations gives the order of the AR model. By the “order of the model” we mean the most extreme lag of x that is used as a predictor. Identification of an MA model is often best done with the ACF rather than the PACF. For an MA model, the theoretical PACF does not shut off, but instead tapers toward 0 in some manner. A clearer pattern for an MA model is in the ACF. The ACF will have non-zero autocorrelations only at lags involved in the model.

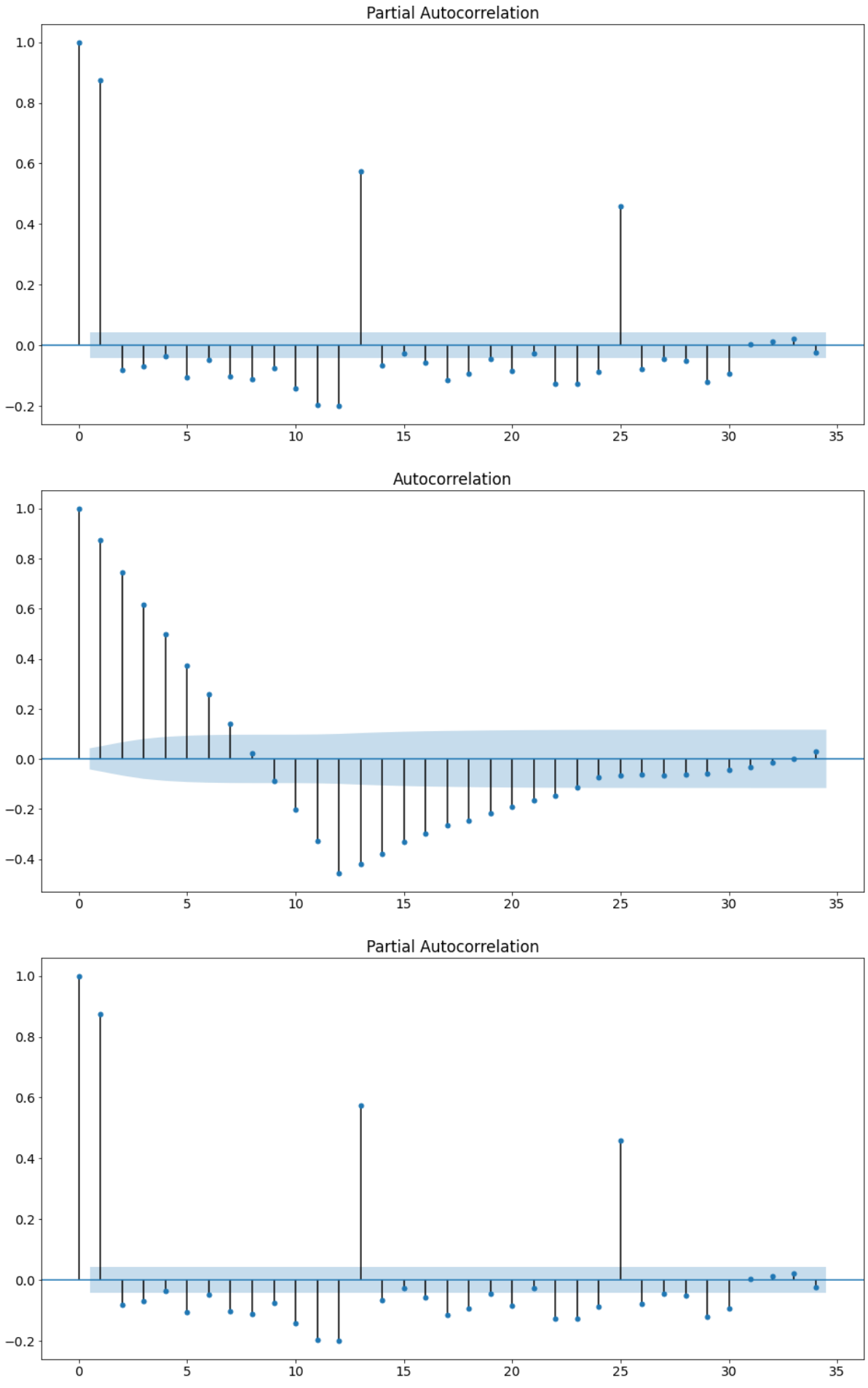
In [235]:
```
# Register converters to avoid warnings
pd.plotting.register_matplotlib_converters()
plt.rc("figure", figsize=(16,8))
plt.rc("font", size=14)
```

Final ACF and PACF Plots

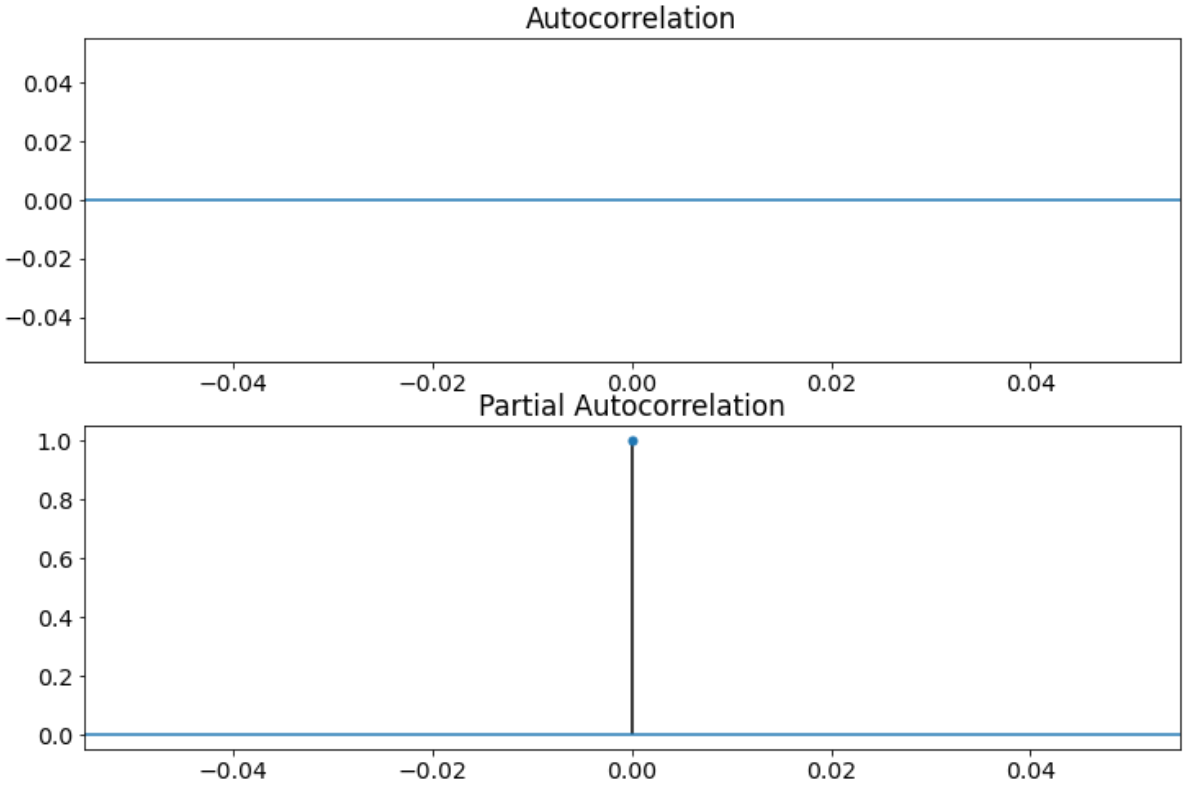
We've run quite a few plots, so let's just quickly get our "final" ACF and PACF plots. These are the ones we will be referencing in the rest of the notebook below.

In [236]:
```
plot_acf(df2['Seasonal First Difference'].dropna())
plot_pacf(df2['Seasonal First Difference'].dropna())
```

Out[236]:


Partial Autocorrelation
Autocorrelation
Partial Autocorrelation

In [237]:
```
fig = plt.figure(figsize=(12,8))
ax1 = fig.add_subplot(211)
fig = sm.graphics.tsa.plot_acf(df2['Seasonal First Difference'].iloc[13:],lags
=40,ax=ax1)
ax2 = fig.add_subplot(212)
fig = sm.graphics.tsa.plot_pacf(df2['Seasonal First Difference'].iloc[13:],lag
s=40,ax=ax2)
```



Using the Seasonal ARIMA model

Finally we can use our ARIMA model now that we have an understanding of our data!

Step 6: Construct the Arima Model

In [238]:
```
# For non-seasonal data
from statsmodels.tsa.arima_model import ARIMA
```

In [239]:
```
help(ARIMA)
```

```
Help on class ARIMA in module statsmodels.tsa.arima_model:

class ARIMA(ARMA)
 |  ARIMA(endog, order, exog=None, dates=None, freq=None, missing='none')
 |
 |  Autoregressive Integrated Moving Average ARIMA(p,d,q) Model
 |
 |  Parameters
 |  ----------
 |  endog : array_like
 |      The endogenous variable.
 |  order : iterable
 |      The (p,d,q) order of the model for the number of AR parameters,
 |      differences, and MA parameters to use.
 |  exog : array_like, optional
 |      An optional array of exogenous variables. This should *not* include a
 |      constant or trend. You can specify this in the `fit` method.
 |  dates : array_like, optional
 |      An array-like object of datetime objects. If a pandas object is given
 |      for endog or exog, it is assumed to have a DateIndex.
 |  freq : str, optional
 |      The frequency of the time-series. A Pandas offset or 'B', 'D', 'W',
 |      'M', 'A', or 'Q'. This is optional if dates are given.
 |
 |
 |  Notes
 |  -----
 |  If exogenous variables are given, then the model that is fit is
 |
 |  .. math::
 |
 |     \phi(L)(y_t - X_t\beta) = \theta(L)\epsilon_t
 |
 |  where :math:`\phi` and :math:`\theta` are polynomials in the lag
 |  operator, :math:`L`. This is the regression model with ARMA errors,
 |  or ARMAX model. This specification is used, whether or not the model
 |  is fit using conditional sum of square or maximum-likelihood, using
 |  the `method` argument in
 |  :meth:`statsmodels.tsa.arima_model.ARIMA.fit`. Therefore, for
 |  now, `css` and `mle` refer to estimation methods only. This may
 |  change for the case of the `css` model in future versions.
 |
 |  Method resolution order:
 |      ARIMA
 |      ARMA
 |      statsmodels.tsa.base.tsa_model.TimeSeriesModel
 |      statsmodels.base.model.LikelihoodModel
 |      statsmodels.base.model.Model
 |      builtins.object
 |
 |  Methods defined here:
 |
 |  __getnewargs__(self)
 |
 |  __init__(self, endog, order, exog=None, dates=None, freq=None, missing='n
one')
 |      Initialize self.  See help(type(self)) for accurate signature.
```

```
 |
 |  fit(self, start_params=None, trend='c', method='css-mle', transparams=Tru
e, solver='lbfgs', maxiter=500, full_output=1, disp=5, callback=None, start_a
r_lags=None, **kwargs)
 |      Fits ARIMA(p,d,q) model by exact maximum likelihood via Kalman filte
r.
 |
 |      Parameters
 |      ----------
 |      start_params : array_like, optional
 |          Starting parameters for ARMA(p,q).  If None, the default is given
 |          by ARMA._fit_start_params.  See there for more information.
 |      transparams : bool, optional
 |          Whether or not to transform the parameters to ensure stationarit
y.
 |          Uses the transformation suggested in Jones (1980).  If False,
 |          no checking for stationarity or invertibility is done.
 |      method : str {'css-mle','mle','css'}
 |          This is the loglikelihood to maximize.  If "css-mle", the
 |          conditional sum of squares likelihood is maximized and its values
 |          are used as starting values for the computation of the exact
 |          likelihood via the Kalman filter.  If "mle", the exact likelihood
 |          is maximized via the Kalman Filter.  If "css" the conditional sum
 |          of squares likelihood is maximized.  All three methods use
 |          `start_params` as starting parameters.  See above for more
 |          information.
 |      trend : str {'c','nc'}
 |          Whether to include a constant or not.  'c' includes constant,
 |          'nc' no constant.
 |      solver : str or None, optional
 |          Solver to be used.  The default is 'lbfgs' (limited memory
 |          Broyden-Fletcher-Goldfarb-Shanno).  Other choices are 'bfgs',
 |          'newton' (Newton-Raphson), 'nm' (Nelder-Mead), 'cg' -
 |          (conjugate gradient), 'ncg' (non-conjugate gradient), and
 |          'powell'. By default, the limited memory BFGS uses m=12 to
 |          approximate the Hessian, projected gradient tolerance of 1e-8 and
 |          factr = 1e2. You can change these by using kwargs.
 |      maxiter : int, optional
 |          The maximum number of function evaluations. Default is 500.
 |      tol : float
 |          The convergence tolerance.  Default is 1e-08.
 |      full_output : bool, optional
 |          If True, all output from solver will be available in
 |          the Results object's mle_retvals attribute.  Output is dependent
 |          on the solver.  See Notes for more information.
 |      disp : int, optional
 |          If True, convergence information is printed.  For the default
 |          l_bfgs_b solver, disp controls the frequency of the output during
 |          the iterations. disp < 0 means no output in this case.
 |      callback : function, optional
 |          Called after each iteration as callback(xk) where xk is the curre
nt
 |          parameter vector.
 |      start_ar_lags : int, optional
 |          Parameter for fitting start_params. When fitting start_params,
 |          residuals are obtained from an AR fit, then an ARMA(p,q) model is
 |          fit via OLS using these residuals. If start_ar_lags is None, fit
```

```
 |          an AR process according to best BIC. If start_ar_lags is not Non
e,
 |          fits an AR process with a lag length equal to start_ar_lags.
 |          See ARMA._fit_start_params_hr for more information.
 |      **kwargs
 |          See Notes for keyword arguments that can be passed to fit.
 |
 |      Returns
 |      -------
 |      `statsmodels.tsa.arima.ARIMAResults` class
 |
 |      See Also
 |      --------
 |      statsmodels.base.model.LikelihoodModel.fit : for more information
 |          on using the solvers.
 |      ARIMAResults : results class returned by fit
 |
 |      Notes
 |      -----
 |      If fit by 'mle', it is assumed for the Kalman Filter that the initial
 |      unknown state is zero, and that the initial variance is
 |      P = dot(inv(identity(m**2)-kron(T,T)),dot(R,R.T).ravel('F')).reshape
(r,
 |      r, order = 'F')
 |
 |  predict(self, params, start=None, end=None, exog=None, typ='linear', dyna
mic=False)
 |      ARIMA model in-sample and out-of-sample prediction
 |
 |      Parameters
 |      ----------
 |      params : array_like
 |          The fitted parameters of the model.
 |      start : int, str, or datetime
 |          Zero-indexed observation number at which to start forecasting, i
e.,
 |          the first forecast is start. Can also be a date string to
 |          parse or a datetime type.
 |      end : int, str, or datetime
 |          Zero-indexed observation number at which to end forecasting, ie.,
 |          the first forecast is start. Can also be a date string to
 |          parse or a datetime type. However, if the dates index does not
 |          have a fixed frequency, end must be an integer index if you
 |          want out of sample prediction.
 |      exog : array_like, optional
 |          If the model is an ARMAX and out-of-sample forecasting is
 |          requested, exog must be given. exog must be aligned so that exog
[0]
 |          is used to produce the first out-of-sample forecast. The number o
f
 |          observation in exog should match the number of out-of-sample
 |          forecasts produced. If the length of exog does not match the numb
er
 |          of forecasts, a SpecificationWarning is produced.
 |      dynamic : bool, optional
 |          The `dynamic` keyword affects in-sample prediction. If dynamic
 |          is False, then the in-sample lagged values are used for
```

```
|          prediction. If `dynamic` is True, then in-sample forecasts are
|          used in place of lagged dependent variables. The first forecast
|          value is `start`.
|      typ : str {'linear', 'levels'}
|
|          - 'linear' : Linear prediction in terms of the differenced
|            endogenous variables.
|          - 'levels' : Predict the levels of the original endogenous
|            variables.
|
|
|      Returns
|      -------
|      predict : ndarray
|          The predicted values.
|
|
|
|      Notes
|      -----
|      Use the results predict method instead.
|
|  ----------------------------------------------------------------------
|  Static methods defined here:
|
|  __new__(cls, endog, order, exog=None, dates=None, freq=None, missing='non
e')
|      Create and return a new object.  See help(type) for accurate signatur
e.
|
|  ----------------------------------------------------------------------
|  Methods inherited from ARMA:
|
|  geterrors(self, params)
|      Get the errors of the ARMA process.
|
|      Parameters
|      ----------
|      params : array_like
|          The fitted ARMA parameters
|      order : array_like
|          3 item iterable, with the number of AR, MA, and exogenous
|          parameters, including the trend
|
|  hessian(self, params)
|      Compute the Hessian at params,
|
|      Notes
|      -----
|      This is a numerical approximation.
|
|  loglike(self, params, set_sigma2=True)
|      Compute the log-likelihood for ARMA(p,q) model
|
|      Notes
|      -----
|      Likelihood used depends on the method set in fit
```

```
 |
 |  loglike_css(self, params, set_sigma2=True)
 |      Conditional Sum of Squares likelihood function.
 |
 |  loglike_kalman(self, params, set_sigma2=True)
 |      Compute exact loglikelihood for ARMA(p,q) model by the Kalman Filter.
 |
 |  score(self, params)
 |      Compute the score function at params.
 |
 |      Notes
 |      -----
 |      This is a numerical approximation.
 |
 |  ----------------------------------------------------------------------
 |  Class methods inherited from ARMA:
 |
 |  from_formula(formula, data, subset=None, drop_cols=None, *args, **kwargs)
from builtins.type
 |      Create a Model from a formula and dataframe.
 |
 |      Parameters
 |      ----------
 |      formula : str or generic Formula object
 |          The formula specifying the model.
 |      data : array_like
 |          The data for the model. See Notes.
 |      subset : array_like
 |          An array-like object of booleans, integers, or index values that
 |          indicate the subset of df to use in the model. Assumes df is a
 |          `pandas.DataFrame`.
 |      drop_cols : array_like
 |          Columns to drop from the design matrix.  Cannot be used to
 |          drop terms involving categoricals.
 |      *args
 |          Additional positional argument that are passed to the model.
 |      **kwargs
 |          These are passed to the model with one exception. The
 |          ``eval_env`` keyword is passed to patsy. It can be either a
 |          :class:`patsy:patsy.EvalEnvironment` object or an integer
 |          indicating the depth of the namespace to use. For example, the
 |          default ``eval_env=0`` uses the calling namespace. If you wish
 |          to use a "clean" environment set ``eval_env=-1``.
 |
 |      Returns
 |      -------
 |      model
 |          The model instance.
 |
 |      Notes
 |      -----
 |      data must define __getitem__ with the keys in the formula terms
 |      args and kwargs are passed on to the model instantiation. E.g.,
 |      a numpy structured or rec array, a dictionary, or a pandas DataFrame.
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors inherited from statsmodels.tsa.base.tsa_model.TimeSeries
```

```
Model:
 |
 |  exog_names
 |      The names of the exogenous variables.
 |
 |  ----------------------------------------------------------------------
 |  Methods inherited from statsmodels.base.model.LikelihoodModel:
 |
 |  information(self, params)
 |      Fisher information matrix of model.
 |
 |      Returns -1 * Hessian of the log-likelihood evaluated at params.
 |
 |      Parameters
 |      ----------
 |      params : ndarray
 |          The model parameters.
 |
 |  initialize(self)
 |      Initialize (possibly re-initialize) a Model instance.
 |
 |      For example, if the the design matrix of a linear model changes then
 |      initialized can be used to recompute values using the modified design
 |      matrix.
 |
 |  ----------------------------------------------------------------------
 |  Readonly properties inherited from statsmodels.base.model.Model:
 |
 |  endog_names
 |      Names of endogenous variables.
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors inherited from statsmodels.base.model.Model:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)
```

p,d,q parameters

p: The number of lag observations included in the model. d: The number of times that the raw observations are differenced, also called the degree of differencing. q: The size of the moving average window, also called the order of moving average

Rules to get p,d,q

https://people.duke.edu/~rnau/arimrule.htm (https://people.duke.edu/~rnau/arimrule.htm)

In [240]:
```
df2['AAPL']
```

Out[240]:
```
0         7.40
1         7.45
2         7.45
3         7.43
4         7.28
          ...
2209    388.81
2210    400.29
2211    402.19
2212    408.43
2213    422.00
Name: AAPL, Length: 2214, dtype: float64
```

In [241]:
```
# df2.index = pd.DatetimeIndex(df2.index).to_period('M')
```

SARIMAX Help

https://www.statsmodels.org/stable/examples/notebooks/generated/statespace_sarimax_stata.html (https://www.statsmodels.org/stable/examples/notebooks/generated/statespace_sarimax_stata.html)

In [242]:
```
# We have seasonal data!
model = sm.tsa.statespace.SARIMAX(df2['AAPL'],order=(0,1,0), seasonal_order=(1
,1,1,12))
```

In [243]:
```
results = model.fit()
print(results.summary())
```

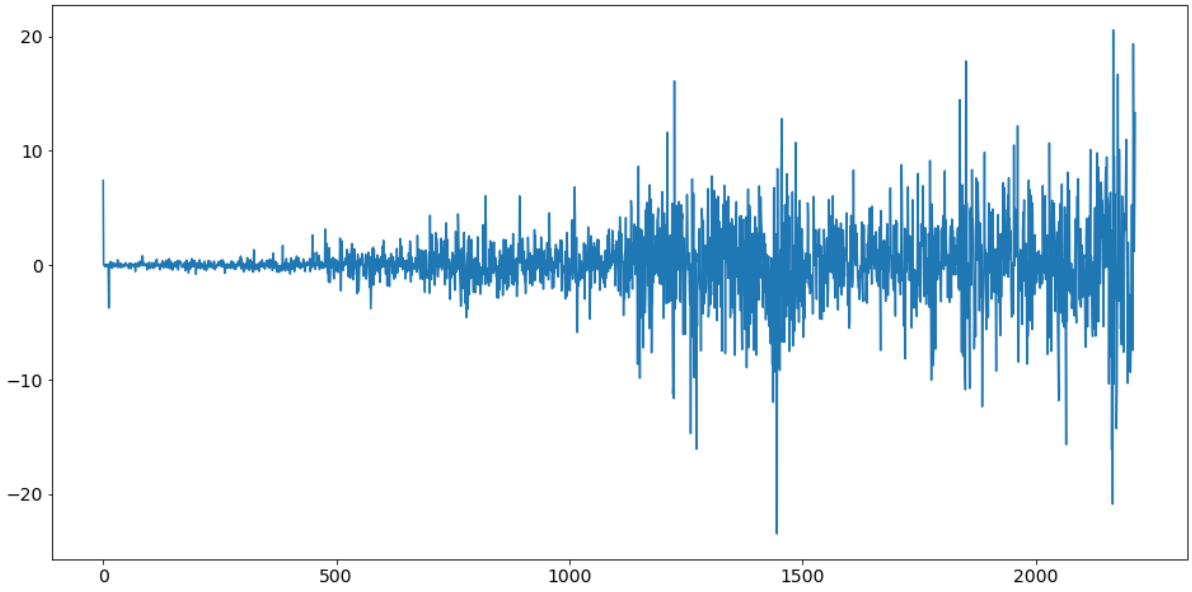
```
                                     SARIMAX Results
==========================================================================================
Dep. Variable:                                AAPL   No. Observations:
2214
Model:             SARIMAX(0, 1, 0)x(1, 1, [1], 12)   Log Likelihood
-5729.291
Date:                              Wed, 11 Nov 2020   AIC
11464.582
Time:                                      08:01:00   BIC
11481.672
Sample:                                           0   HQIC
11470.827
                                              - 2214
Covariance Type:                                opg
==============================================================================
=
                 coef    std err          z      P>|z|      [0.025      0.97
5]
------------------------------------------------------------------------------
-
ar.S.L12       0.0146      0.015      0.954      0.340      -0.015       0.04
5
ma.S.L12      -0.9906      0.006   -158.832      0.000      -1.003      -0.97
8
sigma2        10.4523      0.153     68.150      0.000      10.152      10.75
3
==========================================================================================
======
Ljung-Box (Q):                       58.77   Jarque-Bera (JB):              4
237.99
Prob(Q):                              0.03   Prob(JB):
0.00
Heteroskedasticity (H):              37.05   Skew:
-0.17
Prob(H) (two-sided):                  0.00   Kurtosis:
9.79
==========================================================================================
======

Warnings:
[1] Covariance matrix calculated using the outer product of gradients (comple
x-step).
```

In [244]:
```
results.resid
```

Out[244]:
```
0        7.400000
1        0.050000
2        0.000000
3       -0.020000
4       -0.150000
           ...
2209    19.344712
2210    10.998229
2211     1.199409
2212     5.714726
2213    13.317273
Length: 2214, dtype: float64
```

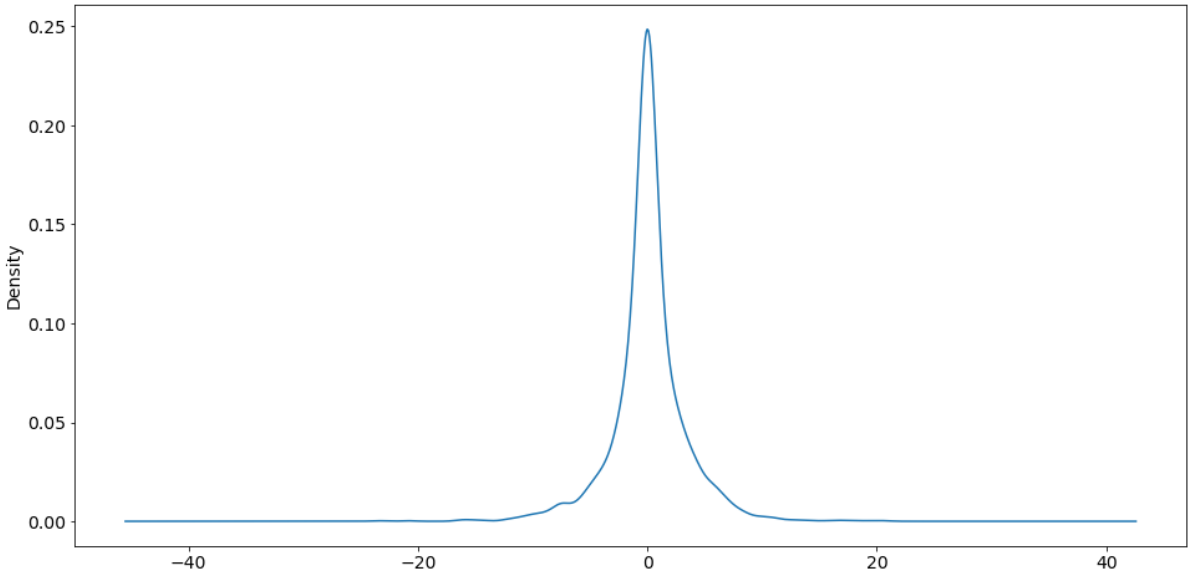
In [245]:
```
results.resid.plot()
```

Out[245]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x7ff4b11791f0>
```



In [246]:
```
# DEnsity Distribution
results.resid.plot(kind='kde')
```

Out[246]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x7ff4b0c628e0>
```



Step 6: Use the model to make predictions

Prediction of Future Values

Firts we can get an idea of how well our model performs by just predicting for values that we actually already know:

In [247]:
```
#Set of columns
df2.columns
```

Out[247]:
```
Index(['Dates', 'AAPL', 'MSFT', 'XOM', 'SPX', 'AAPL First Difference',
       'AAPL Second Difference', 'Seasonal Difference',
       'Seasonal First Difference'],
      dtype='object')
```

In [248]:
```
# Examine the shape of the DataFrame (again)
print(df2.shape)
```

```
(2214, 9)
```

In [249]:
```
df2['forecast'] = results.predict(start =2000,end=9999,dynamic=False)
```

In [250]:
```
df2.tail()
```

Out[250]:

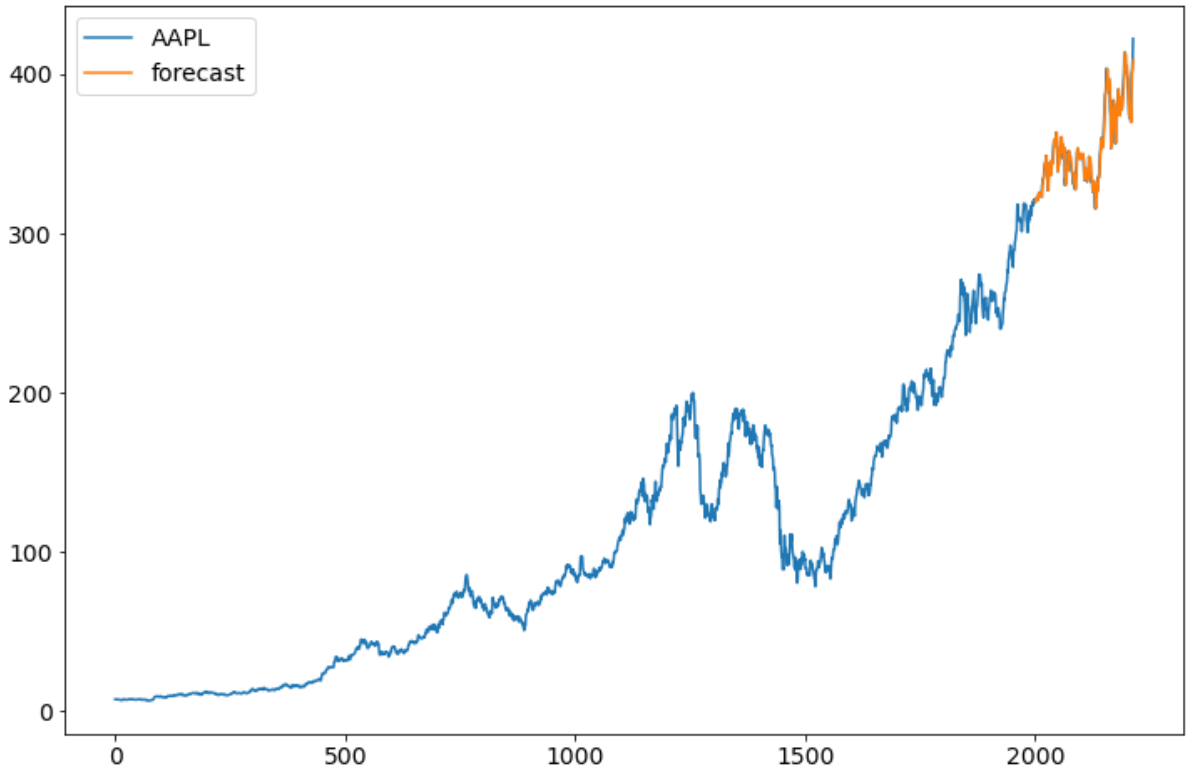
	Dates	AAPL	MSFT	XOM	SPX	AAPL First Difference	AAPL Second Difference	Seasonal Difference	Seasonal First Difference	fore
2209	2011-10-10	388.81	26.94	76.28	1194.89	19.01	26.58	-13.01	-35.09	369.46
2210	2011-10-11	400.29	27.00	76.27	1195.54	11.48	-7.53	-4.01	-24.38	389.29
2211	2011-10-12	402.19	26.96	77.16	1207.25	1.90	-9.58	-0.98	-20.01	400.99
2212	2011-10-13	408.43	27.18	76.37	1203.66	6.24	4.34	9.17	-12.61	402.71
2213	2011-10-14	422.00	27.27	78.11	1224.58	13.57	7.33	24.99	7.92	408.68

In [251]:
```
df2.columns
```

Out[251]:
```
Index(['Dates', 'AAPL', 'MSFT', 'XOM', 'SPX', 'AAPL First Difference',
       'AAPL Second Difference', 'Seasonal Difference',
       'Seasonal First Difference', 'forecast'],
      dtype='object')
```

In [252]:
```
df2[['AAPL','forecast']].plot(figsize=(12,8))
```

Out[252]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x7ff4d0fd2760>
```



Forecasting

This requires more time periods, so let's create them with pandas onto our original dataframe!

In [253]:
```
df2.head()
```

Out[253]:

	Dates	AAPL	MSFT	XOM	SPX	AAPL First Difference	AAPL Second Difference	Seasonal Difference	Seasonal First Difference	forecast
0	2003-01-02	7.40	21.11	29.22	909.03	NaN	NaN	NaN	NaN	NaN
1	2003-01-03	7.45	21.14	29.24	908.59	0.05	NaN	NaN	NaN	NaN
2	2003-01-06	7.45	21.52	29.96	929.01	0.00	-0.05	NaN	NaN	NaN
3	2003-01-07	7.43	21.93	28.95	922.93	-0.02	-0.02	NaN	NaN	NaN
4	2003-01-08	7.28	21.31	28.83	909.93	-0.15	-0.13	NaN	NaN	NaN

In [254]:
```
# https://pandas.pydata.org/pandas-docs/stable/timeseries.html
# Alternatives
# pd.date_range(df.index[-1],periods=12,freq='M')
```

In [255]:
```
df2.tail()
```

Out[255]:

	Dates	AAPL	MSFT	XOM	SPX	AAPL First Difference	AAPL Second Difference	Seasonal Difference	Seasonal First Difference	fore
2209	2011-10-10	388.81	26.94	76.28	1194.89	19.01	26.58	-13.01	-35.09	369.46
2210	2011-10-11	400.29	27.00	76.27	1195.54	11.48	-7.53	-4.01	-24.38	389.29
2211	2011-10-12	402.19	26.96	77.16	1207.25	1.90	-9.58	-0.98	-20.01	400.99
2212	2011-10-13	408.43	27.18	76.37	1203.66	6.24	4.34	9.17	-12.61	402.71
2213	2011-10-14	422.00	27.27	78.11	1224.58	13.57	7.33	24.99	7.92	408.68

In [256]:
```
from pandas.tseries.offsets import DateOffset
```

In [257]:
```
future_dates = pd.date_range(df2.index[-1],periods=12,freq='M')
```

In [258]:
```
#future_dates = [df2.index[-1] + DateOffset(months=x) for x in range(0,24) ]
```

In [259]:
```
future_dates
```

Out[259]:
```
DatetimeIndex(['1970-01-31 00:00:00.000002213',
               '1970-02-28 00:00:00.000002213',
               '1970-03-31 00:00:00.000002213',
               '1970-04-30 00:00:00.000002213',
               '1970-05-31 00:00:00.000002213',
               '1970-06-30 00:00:00.000002213',
               '1970-07-31 00:00:00.000002213',
               '1970-08-31 00:00:00.000002213',
               '1970-09-30 00:00:00.000002213',
               '1970-10-31 00:00:00.000002213',
               '1970-11-30 00:00:00.000002213',
               '1970-12-31 00:00:00.000002213'],
              dtype='datetime64[ns]', freq='M')
```

In [260]:
```
final_df2= pd.concat([df2.index,future_dates])
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-260-6eab0ca97ab7> in <module>
----> 1 final_df2= pd.concat([df2.index,future_dates])

~/anaconda3/lib/python3.8/site-packages/pandas/core/reshape/concat.py in conc
at(objs, axis, join, ignore_index, keys, levels, names, verify_integrity, sor
t, copy)
    269     ValueError: Indexes have overlapping values: ['a']
    270     """
--> 271     op = _Concatenator(
    272         objs,
    273         axis=axis,

~/anaconda3/lib/python3.8/site-packages/pandas/core/reshape/concat.py in __in
it__(self, objs, axis, join, keys, levels, names, ignore_index, verify_integr
ity, copy, sort)
    355                     "only Series and DataFrame objs are valid".format
(typ=type(obj))
    356                 )
--> 357                 raise TypeError(msg)
    358
    359             # consolidate

TypeError: cannot concatenate object of type '<class 'pandas.core.indexes.ran
ge.RangeIndex'>'; only Series and DataFrame objs are valid
```

In [261]:
```
final_df2 = pd.DataFrame(index=future_dates,columns=df2.columns)
```

In [262]:
```
final_df2.head()
```

Out[262]:

	Dates	AAPL	MSFT	XOM	SPX	AAPL First Difference	AAPL Second Difference	Seasonal Difference	Season Firs Differenc
1970-01-31 00:00:00.000002213	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	Na
1970-02-28 00:00:00.000002213	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	Na
1970-03-31 00:00:00.000002213	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	Na
1970-04-30 00:00:00.000002213	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	Na
1970-05-31 00:00:00.000002213	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	Na

In [263]:
```
final_df2.tail()
```

Out[263]:

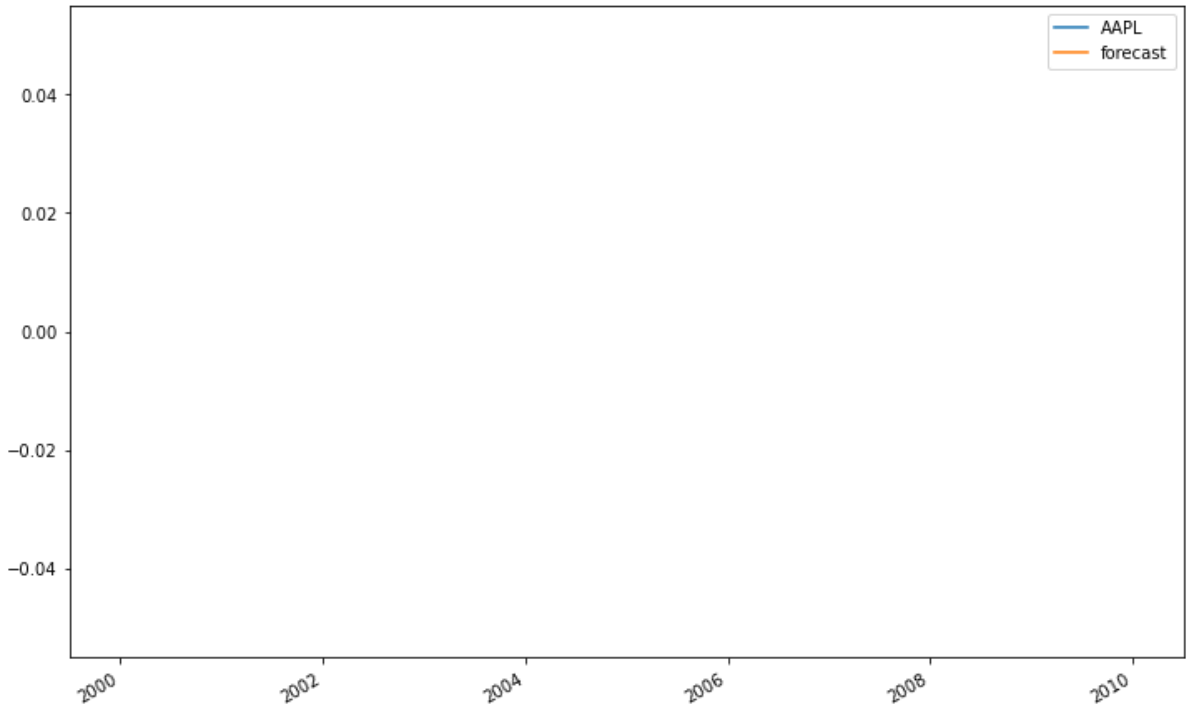
	Dates	AAPL	MSFT	XOM	SPX	AAPL First Difference	AAPL Second Difference	Seasonal Difference	Season Firs Differenc
1970-08-31 00:00:00.000002213	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	Na
1970-09-30 00:00:00.000002213	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	Na
1970-10-31 00:00:00.000002213	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	Na
1970-11-30 00:00:00.000002213	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	Na
1970-12-31 00:00:00.000002213	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	Na

In [264]:
```
print(future_df2.shape)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-264-9c73179b3872> in <module>
----> 1 print(future_df2.shape)

NameError: name 'future_df2' is not defined
```

In [275]:
```
final_df2['forecast']= results.predict(start =2000, end = 9999, dynamic= True)
final_df2[['AAPL', 'forecast']].plot(figsize=(12, 8))
```

Out[275]: <matplotlib.axes._subplots.AxesSubplot at 0x7ff498c42850>



In [ ]:

PLOTLY, CUFFLINKS Stock Risk Analysis

In [266]:
```
#install cufflinks via anaconda prompt using these commands - pip install plot
ly & pip install cufflinks
#install library packages
import plotly
import cufflinks as cf
import pandas as pd
import numpy as np
#Enabling the offline mode for interactive plotting locally
from plotly.offline import download_plotlyjs,init_notebook_mode,plot,iplot
init_notebook_mode(connected=True)
cf.go_offline()
#To display the plots
%matplotlib inline
```

In [267]:
```
# Get, check and clean the Data for analysis
df_G7=pd.read_csv('stock_px_2.csv')
```

In [268]:
```
# check the head of DataFrame df
df_G7.head()
```

Out[268]:

	Unnamed: 0	AAPL	MSFT	XOM	SPX
0	2003-01-02 00:00:00	7.40	21.11	29.22	909.03
1	2003-01-03 00:00:00	7.45	21.14	29.24	908.59
2	2003-01-06 00:00:00	7.45	21.52	29.96	929.01
3	2003-01-07 00:00:00	7.43	21.93	28.95	922.93
4	2003-01-08 00:00:00	7.28	21.31	28.83	909.93

In [269]:
```
# B_1: Get rid of the coulmn name 'unnamed:0'
df_G7 = pd.read_csv('stock_px_2.csv')
df_G7.rename(columns=({'Unnamed: 0':'Dates'}))
```

Out[269]:

	Dates	AAPL	MSFT	XOM	SPX
0	2003-01-02 00:00:00	7.40	21.11	29.22	909.03
1	2003-01-03 00:00:00	7.45	21.14	29.24	908.59
2	2003-01-06 00:00:00	7.45	21.52	29.96	929.01
3	2003-01-07 00:00:00	7.43	21.93	28.95	922.93
4	2003-01-08 00:00:00	7.28	21.31	28.83	909.93
...	...	...	...	...	...
2209	2011-10-10 00:00:00	388.81	26.94	76.28	1194.89
2210	2011-10-11 00:00:00	400.29	27.00	76.27	1195.54
2211	2011-10-12 00:00:00	402.19	26.96	77.16	1207.25
2212	2011-10-13 00:00:00	408.43	27.18	76.37	1203.66
2213	2011-10-14 00:00:00	422.00	27.27	78.11	1224.58

2214 rows × 5 columns

In [270]:
```
# B_2: Get rid of the coulmn name 'unnamed:0'
df_G8=df_G7.rename(columns=({'Unnamed: 0':'Dates'}))
```

In [271]:
```
# check the head of DataFrame df
df_G8.head()
```

Out[271]:

	Dates	AAPL	MSFT	XOM	SPX
0	2003-01-02 00:00:00	7.40	21.11	29.22	909.03
1	2003-01-03 00:00:00	7.45	21.14	29.24	908.59
2	2003-01-06 00:00:00	7.45	21.52	29.96	929.01
3	2003-01-07 00:00:00	7.43	21.93	28.95	922.93
4	2003-01-08 00:00:00	7.28	21.31	28.83	909.93

In [272]:
```
#Download dataset for analysis
Stock_1 = df_G8
#Call dataset
Stock_1
```

Out[272]:

	Dates	AAPL	MSFT	XOM	SPX
0	2003-01-02 00:00:00	7.40	21.11	29.22	909.03
1	2003-01-03 00:00:00	7.45	21.14	29.24	908.59
2	2003-01-06 00:00:00	7.45	21.52	29.96	929.01
3	2003-01-07 00:00:00	7.43	21.93	28.95	922.93
4	2003-01-08 00:00:00	7.28	21.31	28.83	909.93
...	...	...	...	...	...
2209	2011-10-10 00:00:00	388.81	26.94	76.28	1194.89
2210	2011-10-11 00:00:00	400.29	27.00	76.27	1195.54
2211	2011-10-12 00:00:00	402.19	26.96	77.16	1207.25
2212	2011-10-13 00:00:00	408.43	27.18	76.37	1203.66
2213	2011-10-14 00:00:00	422.00	27.27	78.11	1224.58

2214 rows × 5 columns

In [273]:
```
# check the index of the columns
Stock_1.columns
```

Out[273]:
```
Index(['Dates', 'AAPL', 'MSFT', 'XOM', 'SPX'], dtype='object')
```

In [274]:
```
#Selecting columns to display in a "simple plot"
Stock_1[['AAPL', 'MSFT', 'XOM', 'SPX']].iplot()
```

Appendice 1: Help ARIMA

Help on class ARIMA in module statsmodels.tsa.arima_model:

class ARIMA(ARMA) | Autoregressive Integrated Moving Average ARIMA(p,d,q) Model |
| Parameters | ---------- | endog : array-like | The endogenous variable. | order : iterable | The (p,d,q) order of the model for the number of AR parameters, | differences, and MA parameters to use. | exog : array-like, optional | An optional array of exogenous variables. This should *not* include a | constant or trend. You can specify this in the `fit` method. | dates : array-like of datetime, optional | An array-like object of datetime objects. If a pandas object is given | for endog or exog, it is assumed to have a DateIndex. | freq : str, optional | The frequency of the time-series. A Pandas offset or 'B', 'D', 'W', | 'M', 'A', or 'Q'. This is optional if dates are given. |
|
| Notes | ----- | If exogenous variables are given, then the model that is fit is |
| .. math:: |
| \phi(L)(y_t - X_t\beta) = \theta(L)\epsilon_t |
| where :math: `\phi` and :math: `\theta` are polynomials in the lag | operator, :math: `L` . This is the regression model with ARMA errors, | or ARMAX model. This specification is used, whether or not the model | is fit using conditional sum of square or maximum-likelihood, using | the `method` argument in |
:meth: `statsmodels.tsa.arima_model.ARIMA.fit` . Therefore, for | now, `css` and `mle` refer to estimation methods only. This may | change for the case of the `css` model in future versions. |
| Method resolution order: | ARIMA | ARMA | statsmodels.tsa.base.tsa_model.TimeSeriesModel |
statsmodels.base.model.LikelihoodModel | statsmodels.base.model.Model | builtins.object |
| Methods defined here: |
| **getnewargs**(self) |
| **init**(self, endog, order, exog=None, dates=None, freq=None, missing='none') | Initialize self. See help(type(self)) for accurate signature. |
| fit(self, start_params=None, trend='c', method='css-mle', transparams=True, solver='lbfgs', maxiter=50, full_output=1, disp=5, callback=None, start_ar_lags=None, **kwargs) | **Fits ARIMA(p,d,q) model by exact maximum likelihood via Kalman filter. |**
| Parameters | ---------- | start_params : array-like, optional | Starting parameters for ARMA(p,q). If None, the default is given | by ARMA._fit_start_params. See there for more information. | transparams : bool, optional | Whehter or not to transform the parameters to ensure stationarity. | Uses the transformation suggested in Jones (1980). If False, | no checking for stationarity or invertibility is done. | method : str {'css-mle','mle','css'} | This is the loglikelihood to maximize. If "css-mle", the | conditional sum of squares likelihood is maximized and its values | are used as starting values for the computation of the exact | likelihood via the Kalman filter. If "mle", the exact likelihood | is maximized via the Kalman Filter. If "css" the conditional sum | of squares likelihood is maximized. All three methods use | `start_params` as starting parameters. See above for more | information. | trend : str {'c','nc'} | Whether to include a constant or not. 'c' includes constant, | 'nc' no constant. | solver : str or None, optional | Solver to be used. The default is 'lbfgs' (limited memory | Broyden-Fletcher-Goldfarb-Shanno). Other choices are 'bfgs', | 'newton' (Newton-Raphson), 'nm' (Nelder-Mead), 'cg' - | (conjugate gradient), 'ncg' (non-conjugate gradient), and | 'powell'. By default, the limited memory BFGS uses m=12 to | approximate the Hessian, projected gradient tolerance of 1e-8 and | factr = 1e2. You can change these by using kwargs. | maxiter : int, optional | The maximum number of function evaluations. Default is 50. | tol : float | The convergence tolerance. Default is 1e-08. | full_output : bool, optional | If True, all output from solver will be available in | the Results object's mle_retvals attribute. Output is dependent | on the solver. See Notes for more information. | disp : int, optional | If True, convergence information is printed. For the default | l_bfgs_b solver, disp controls the frequency of the output during | the iterations. disp < 0 means no output in this case. | callback : function, optional | Called after each iteration as callback(xk) where xk is the current |

parameter vector. | start_ar_lags : int, optional | Parameter for fitting start_params. When fitting start_params, | residuals are obtained from an AR fit, then an ARMA(p,q) model is | fit via OLS using these residuals. If start_ar_lags is None, fit | an AR process according to best BIC. If start_ar_lags is not None, | fits an AR process with a lag length equal to start_ar_lags. | See ARMA._fit_start_params_hr for more information. | kwargs | See Notes for keyword arguments that can be passed to fit. |
| Returns | ------- | `statsmodels.tsa.arima.ARIMAResults` **class |**
| See also | -------- | statsmodels.base.model.LikelihoodModel.fit : for more information | on using the solvers. | ARIMAResults : results class returned by fit |
| Notes | ------ | If fit by 'mle', it is assumed for the Kalman Filter that the initial | unkown state is zero, and that the inital variance is | P = dot(inv(identity(m2)-kron(T,T)),dot(R,R.T).ravel('F')).reshape(r, | r, order = 'F') |
predict(self, params, start=None, end=None, exog=None, typ='linear', dynamic=False)	ARIMA model in-sample and out-of-sample prediction
Parameters	----------	params : array-like	The fitted parameters of the model.	start : int, str, or datetime	Zero-indexed observation number at which to start forecasting, ie.,	the first forecast is start. Can also be a date string to	parse or a datetime type.	end : int, str, or datetime	Zero-indexed observation number at which to end forecasting, ie.,	the first forecast is start. Can also be a date string to	parse or a datetime type. However, if the dates index does not	have a fixed frequency, end must be an integer index if you	want out of sample prediction.	exog : array-like, optional	If the model is an ARMAX and out-of-sample forecasting is	requested, exog must be given. Note that you'll need to pass	`k_ar` additional lags for any exogenous variables. E.g., if you	fit an ARMAX(2, q) model and want to predict 5 steps, you need 7	observations to do this.	dynamic : bool, optional	The `dynamic` keyword affects in-sample prediction. If dynamic	is False, then the in-sample lagged values are used for	prediction. If `dynamic` is True, then in-sample forecasts are	used in place of lagged dependent variables. The first forecasted	value is `start`.	typ : str {'linear', 'levels'}
- 'linear' : Linear prediction in terms of the differenced	endogenous variables.	- 'levels' : Predict the levels of the original endogenous	variables.
Returns	-------	predict : array	The predicted values.
Notes	-----	Use the results predict method instead.
----------	Static methods defined here:
new(cls, endog, order, exog=None, dates=None, freq=None, missing='none')	Create and return a new object. See help(type) for accurate signature.
----------	Methods inherited from ARMA:
geterrors(self, params)	Get the errors of the ARMA process.
Parameters	----------	params : array-like	The fitted ARMA parameters	order : array-like	3 item iterable, with the number of AR, MA, and exogenous	parameters, including the trend
hessian(self, params)	Compute the Hessian at params,
Notes	-----	This is a numerical approximation.
loglike(self, params, set_sigma2=True)	Compute the log-likelihood for ARMA(p,q) model
Notes	-----	Likelihood used depends on the method set in fit
loglike_css(self, params, set_sigma2=True)	Conditional Sum of Squares likelihood function.
loglike_kalman(self, params, set_sigma2=True)	Compute exact loglikelihood for ARMA(p,q) model by the Kalman Filter.
score(self, params)	Compute the score function at params.
Notes	-----	This is a numerical approximation.
----------	Data descriptors inherited from statsmodels.tsa.base.tsa_model.TimeSeriesModel:

| exog_names |
| ---------- | Methods inherited from statsmodels.base.model.LikelihoodModel: |
| information(self, params) | Fisher information matrix of model |
| Returns -Hessian of loglike evaluated at params. |
initialize(self)	Initialize (possibly re-initialize) a Model instance. For	instance, the design matrix of a linear model may change	and some things must be recomputed.
----------	Class methods inherited from statsmodels.base.model.Model:
from_formula(formula, data, subset=None, drop_cols=None, *args, **kwargs) from builtins.type	Create a Model from a formula and dataframe.
Parameters	----------	formula : str or generic Formula object	The formula specifying the model	data : array-like	The data for the model. See Notes.	subset : array-like	An array-like object of booleans, integers, or index values that	indicate the subset of df to use in the model. Assumes df is a	`pandas.DataFrame`	drop_cols : array-like	Columns to drop from the design matrix. Cannot be used to	drop terms involving categoricals.	args : extra arguments	These are passed to the model	kwargs : extra keyword arguments	These are passed to the model with one exception. The	`eval_env` keyword is passed to patsy. It can be either a	:class: `patsy:patsy.EvalEnvironment` object or an integer	indicating the depth of the namespace to use. For example, the	default `eval_env=0` uses the calling namespace. If you wish	to use a "clean" environment set `eval_env=-1`.
Returns	-------	model : Model instance
Notes	------	data must define **getitem** with the keys in the formula terms	args and kwargs are passed on to the model instantiation. E.g.,	a numpy structured or rec array, a dictionary, or a pandas DataFrame.
----------	Data descriptors inherited from statsmodels.base.model.Model:
dict	dictionary for instance variables (if defined)
weakref	list of weak references to the object (if defined)
endog_names	Names of endogenous variables