```
In [ ]:
```

Warning! This is a complicated topic! Remember that this is an optional notebook to go through and that to fully understand it you should read the supplemental links and watch the full explanatory walkthrough video. This notebook and the video lectures are not meant to be a full comprehensive overview of ARIMA, but instead a walkthrough of what you can use it for, so you can later understand why it may or may not be a good choice for Financial Stock Data.

ARIMA and Seasonal ARIMA

Autoregressive Integrated Moving Averages

The general process for ARIMA models is the following:

- 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

Let's go through these steps!

Step 1: Get the Data (and format it)

We will be using some data about monthly milk production, full details on it can be found here.

Its saved as a csv for you already, let's load it up:

In [2]:
```python
import numpy as np
import pandas as pd
import statsmodels.api as sm

import matplotlib.pyplot as plt
%matplotlib inline
```

In [3]:
```python
df = pd.read_csv('monthly-milk-production-pounds-p.csv')
```

In [5]:
```python
df.head()
```

Out[5]:

	Month	Monthly milk production: pounds per cow. Jan 62 ? Dec 75
0	1962-01	589.0
1	1962-02	561.0
2	1962-03	640.0
3	1962-04	656.0
4	1962-05	727.0

In [6]:
```python
df.tail()
```

Out[6]:

	Month	Monthly milk production: pounds per cow. Jan 62 ? Dec 75
164	1975-09	817.0
165	1975-10	827.0
166	1975-11	797.0
167	1975-12	843.0
168	Monthly milk production: pounds per cow. Jan 6...	NaN

Clean Up

Let's clean this up just a little!

In [10]:
```python
df.columns = ['Month','Milk in pounds per cow']
df.head()
```

Out[10]:

	Month	Milk in pounds per cow
0	1962-01	589.0
1	1962-02	561.0
2	1962-03	640.0
3	1962-04	656.0
4	1962-05	727.0

In [11]:
```python
# Weird last value at bottom causing issues
df.drop(168,axis=0,inplace=True)
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
Input In [11], in <cell line: 2>()
      1 # Weird last value at bottom causing issues
----> 2 df.drop(168,axis=0,inplace=True)

File ~\anaconda3\lib\site-packages\pandas\util\_decorators.py:311, in deprecate_nonkeyword_arguments.<locals>.decorate.<locals>.wrapper(*args, **kwargs)
    305 if len(args) > num_allow_args:
    306     warnings.warn(
    307         msg.format(arguments=arguments),
    308         FutureWarning,
    309         stacklevel=stacklevel,
    310     )
--> 311 return func(*args, **kwargs)

File ~\anaconda3\lib\site-packages\pandas\core\frame.py:4954, in DataFrame.drop(self, labels, axis, index, columns, level, inplace, errors)
   4806 @deprecate_nonkeyword_arguments(version=None, allowed_args=["self", "labels"])
   4807 def drop(
   4808     self,
   (...)
   4815     errors: str = "raise",
   4816 ):
   4817     """
   4818     Drop specified labels from rows or columns.
   4819
   (...)
   4952             weight  1.0     0.8
   4953     """
-> 4954     return super().drop(
   4955         labels=labels,
   4956         axis=axis,
   4957         index=index,
   4958         columns=columns,
   4959         level=level,
   4960         inplace=inplace,
   4961         errors=errors,
   4962     )

File ~\anaconda3\lib\site-packages\pandas\core\generic.py:4267, in NDFrame.drop(self, labels, axis, index, columns, level, inplace, errors)
   4265 for axis, labels in axes.items():
   4266     if labels is not None:
-> 4267         obj = obj._drop_axis(labels, axis, level=level, errors=errors)
   4269 if inplace:
   4270     self._update_inplace(obj)

File ~\anaconda3\lib\site-packages\pandas\core\generic.py:4311, in NDFrame._drop_axis(self, labels, axis, level, errors, consolidate, only_slice)
   4309         new_axis = axis.drop(labels, level=level, errors=errors)
   4310     else:
-> 4311         new_axis = axis.drop(labels, errors=errors)
   4312     indexer = axis.get_indexer(new_axis)
   4314 # Case for non-unique axis
   4315 else:

File ~\anaconda3\lib\site-packages\pandas\core\indexes\base.py:6644, in Index.drop(self, labels, errors)
   6642 if mask.any():
   6643     if errors != "ignore":
-> 6644         raise KeyError(f"{list(labels[mask])} not found in axis")
   6645     indexer = indexer[~mask]
   6646 return self.delete(indexer)

KeyError: '[168] not found in axis'
```

In [12]:
```python
df['Month'] = pd.to_datetime(df['Month'])
```

In [13]:
```python
df.head()
```

Out[13]:

	Month	Milk in pounds per cow
0	1962-01-01	589.0
1	1962-02-01	561.0
2	1962-03-01	640.0
3	1962-04-01	656.0
4	1962-05-01	727.0

In [14]:
```python
df.set_index('Month',inplace=True)
```

In [15]:
```python
df.head()
```

Out[15]:

Month	Milk in pounds per cow
1962-01-01	589.0
1962-02-01	561.0
1962-03-01	640.0
1962-04-01	656.0
1962-05-01	727.0

In [16]:
```python
df.describe().transpose()
```

Out[16]:

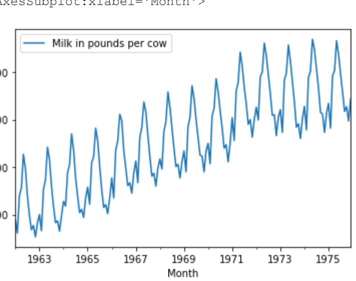
	count	mean	std	min	25%	50%	75%	max
Milk in pounds per cow	168.0	754.708333	102.204524	553.0	677.75	761.0	824.5	969.0

Step 2: Visualize the Data

Let's visualize this data with a few methods.

In [17]:
```python
df.plot()
```

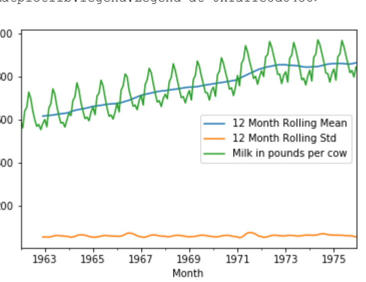
Out[17]:
```
<AxesSubplot:xlabel='Month'>
```



In [18]:
```python
timeseries = df['Milk in pounds per cow']
```

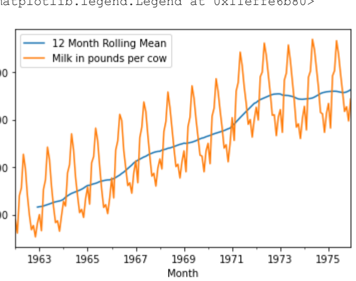
In [14]:
```python
timeseries.rolling(12).mean().plot(label='12 Month Rolling Mean')
timeseries.rolling(12).std().plot(label='12 Month Rolling Std')
timeseries.plot()
plt.legend()
```

Out[14]:
```
<matplotlib.legend.Legend at 0x1a11e5d8438>
```



In [21]:
```python
timeseries.rolling(12).mean().plot(label='12 Month Rolling Mean')
timeseries.plot()
plt.legend()
```

Out[21]:
```
<matplotlib.legend.Legend at 0x11effe6b80>
```



Decomposition

ETS decomposition allows us to see the individual parts!

In [22]:
```python
from statsmodels.tsa.seasonal import seasonal_decompose
decomposition = seasonal_decompose(df['Milk in pounds per cow'], freq=12)
fig = plt.figure()
fig = decomposition.plot()
fig.set_size_inches(15, 8)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Input In [22], in <cell line: 2>()
      1 from statsmodels.tsa.seasonal import seasonal_decompose
----> 2 decomposition = seasonal_decompose(df['Milk in pounds per cow'], freq=12)
      3 fig = plt.figure()
      4 fig = decomposition.plot()

TypeError: seasonal_decompose() got an unexpected keyword argument 'freq'
```

Testing for Stationarity

We can use the Augmented Dickey-Fuller unit root test.

In statistics and econometrics, an augmented Dickey–Fuller test (ADF) tests the null hypothesis that a unit root is present in a time series sample. The alternative hypothesis is different depending on which version of the test is used, but is usually stationarity or trend-stationarity.

Basically, we are trying to whether to accept the Null Hypothesis **H0** (that the time series has a unit root, indicating it is non-stationary) or reject **H0** and go with the Alternative Hypothesis (that the time series has no unit root and is stationary).

We end up deciding this based on the p-value return.

- A small p-value (typically ≤ 0.05) indicates strong evidence against the null hypothesis, so you reject the null hypothesis.
- A large p-value (> 0.05) indicates weak evidence against the null hypothesis, so you fail to reject the null hypothesis.

Let's run the Augmented Dickey-Fuller test on our data:

In [23]:
```python
df.head()
```

Out[23]:

Month	Milk in pounds per cow
1962-01-01	589.0
1962-02-01	561.0
1962-03-01	640.0
1962-04-01	656.0
1962-05-01	727.0

In [24]:
```python
from statsmodels.tsa.stattools import adfuller
```

In [19]:
```python
result = adfuller(df['Milk in pounds per cow'])
```

In [20]:
```python
print('Augmented Dickey-Fuller Test:')
labels = ['ADF Test Statistic','p-value','#Lags Used','Number of Observations 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 hypothesis. Data has no unit root and i
else:
    print("weak evidence against null hypothesis, time series has a unit root, indicating it is non-stationary
```

```
Augmented Dickey-Fuller Test:
ADF Test Statistic : -1.30381158742
p-value : 0.627426708603
#Lags Used : 13
Number of Observations Used : 154
weak evidence against null hypothesis, time series has a unit root, indicating it is non-stationary
```

In [25]:
```python
# 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 Observations 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 hypothesis. Data has no unit root a
    else:
        print("weak evidence against null hypothesis, time series has a unit root, indicating it is non-station
```

Important 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!

Differencing

The first difference of a time series is the series of changes from one period to the next. We can do this easily with pandas. You can continue to take the second difference, third difference, and so on until your data is stationary.

First Difference

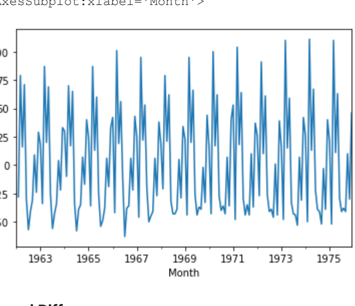
In [26]:
```python
df['Milk First Difference'] = df['Milk in pounds per cow'] - df['Milk in pounds per cow'].shift(1)
```

In [27]:
```python
adf_check(df['Milk First Difference'].dropna())
```

```
Augmented Dickey-Fuller Test:
ADF Test Statistic : -3.0549955586530553
p-value : 0.03006800400178688
#Lags Used : 14
Number of Observations Used : 152
strong evidence against the null hypothesis, reject the null hypothesis. Data has no unit root and is stationar
y
```

In [28]:
```python
df['Milk First Difference'].plot()
```

Out[28]:
```
<AxesSubplot:xlabel='Month'>
```



Second Difference

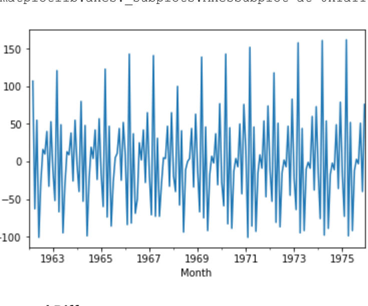
In [29]:
```python
# Sometimes it would be necessary to do a second difference
# This is just for show, we didn't need to do a second difference in our case
df['Milk Second Difference'] = df['Milk First Difference'] - df['Milk First Difference'].shift(1)
```

In [30]:
```python
adf_check(df['Milk Second Difference'].dropna())
```

```
Augmented Dickey-Fuller Test:
ADF Test Statistic : -14.327873645603336
p-value : 1.1126989332083069e-26
#Lags Used : 11
Number of Observations Used : 154
strong evidence against the null hypothesis, reject the null hypothesis. Data has no unit root and is stationar
y
```

In [27]:
```python
df['Milk Second Difference'].plot()
```

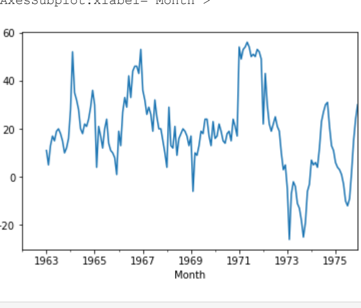
Out[27]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x1a11ea602b0>
```



Seasonal Difference

In [31]:
```python
df['Seasonal Difference'] = df['Milk in pounds per cow'] - df['Milk in pounds per cow'].shift(12)
df['Seasonal Difference'].plot()
```

Out[31]:
```
<AxesSubplot:xlabel='Month'>
```



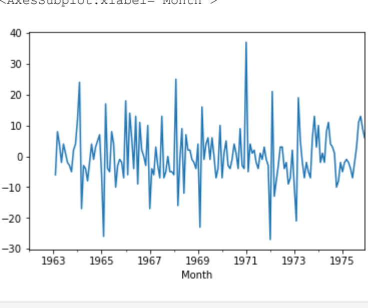
In [32]:
```python
# Seasonal Difference by itself was not enough!
adf_check(df['Seasonal Difference'].dropna())
```

```
Augmented Dickey-Fuller Test:
ADF Test Statistic : -2.3354193143593993
p-value : 0.16079880527711304
#Lags Used : 12
Number of Observations Used : 143
weak evidence against null hypothesis, time series has a unit root, indicating it is non-stationary
```

Seasonal First Difference

In [33]:
```python
# You can also do seasonal first difference
df['Seasonal First Difference'] = df['Milk First Difference'] - df['Milk First Difference'].shift(12)
df['Seasonal First Difference'].plot()
```

Out[33]:
```
<AxesSubplot:xlabel='Month'>
```



In [34]:
```python
adf_check(df['Seasonal First Difference'].dropna())
```

```
Augmented Dickey-Fuller Test:
ADF Test Statistic : -5.038002274921985
p-value : 1.86542343187882e-05
#Lags Used : 11
Number of Observations Used : 143
strong evidence against the null hypothesis, reject the null hypothesis. Data has no unit root and is stationar
y
```

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.

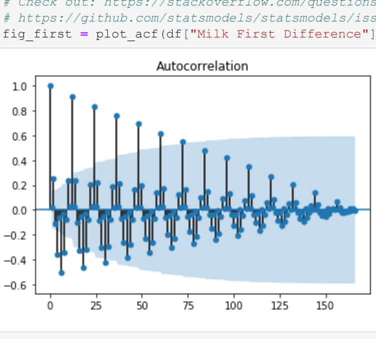
Important Note!

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.

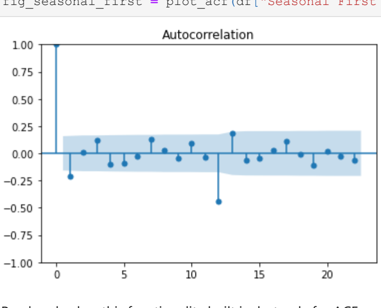
The reason we use two here is to show you the two typical types of behaviour you would see when using ACF.

In [35]:
```python
from statsmodels.graphics.tsaplots import plot_acf,plot_pacf
```

In [33]:
```python
# Duplicate plots
# Check out: https://stackoverflow.com/questions/21788593/statsmodels-duplicate-charts
# https://github.com/statsmodels/statsmodels/issues/1265
fig_first = plot_acf(df["Milk First Difference"].dropna())
```



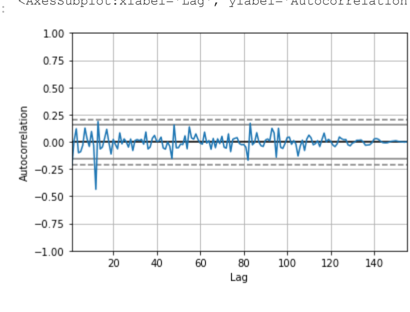
In [36]:
```python
fig_seasonal_first = plot_acf(df["Seasonal First Difference"].dropna())
```



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 [37]:
```python
from pandas.plotting import autocorrelation_plot
autocorrelation_plot(df['Seasonal First Difference'].dropna())
```

Out[37]:
```
<AxesSubplot:xlabel='Lag', ylabel='Autocorrelation'>
```



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.

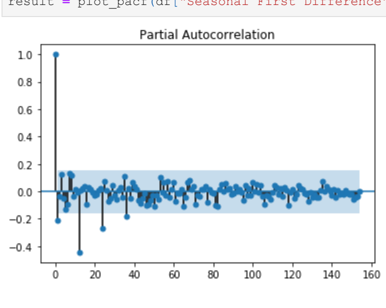
Formally, this is relationship is defined as:

$\frac{\text{Covariance}(y, x_3|x_1, x_2)}{\sqrt{\text{Variance}(y|x_1, x_2)\text{Variance}(x_3| x_1, x_2)}}$

Check out this link for full details on this.

We can then plot this relationship:

In [36]:
```python
result = plot_pacf(df["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

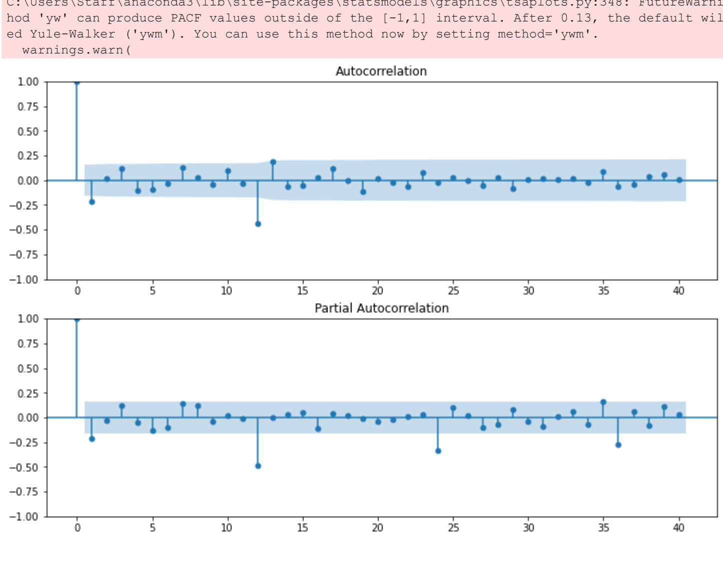
- Identification of an AR model is often best done with the PACF.
  - For an AR model, the theoretical PACF "shuts off" past the order of the model. The phrase "shuts off" means that in theory the partial autocorrelations are equal to 0 beyond that point. Put another way, the number of non-zero partial 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.

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 [38]:
```python
fig = plt.figure(figsize=(12,8))
ax1 = fig.add_subplot(211)
fig = sm.graphics.tsa.plot_acf(df['Seasonal First Difference'].iloc[13:], lags=40, ax=ax1)
ax2 = fig.add_subplot(212)
fig = sm.graphics.tsa.plot_pacf(df['Seasonal First Difference'].iloc[13:], lags=40, ax=ax2)
```

```
C:\Users\Staff\anaconda3\lib\site-packages\statsmodels\graphics\tsaplots.py:348: FutureWarning: The default met
hod 'yw' can produce PACF values outside of the [-1,1] interval. After 0.13, the default will change tounadjust
ed Yule-Walker ('ywm'). You can use this method now by setting method='ywm'.
  warnings.warn(
```



Using the Seasonal ARIMA model

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

In [38]:
```python
# For non-seasonal data
from statsmodels.tsa.arima_model import ARIMA
```

In [39]:
```python
# I recommend you glance over this!

#
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, ful
l_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(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', 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
```

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.

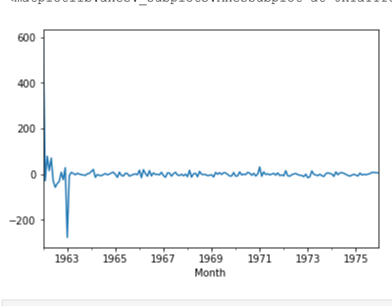
In [55]:
```
# We have seasonal data!
model = sm.tsa.statespace.SARIMAX(df['Milk in pounds per cow'],order=(0,1,0), seasonal_order=(1,1,1,12))
results = model.fit()
print(results.summary())
```

```
                                 Statespace Model Results
==========================================================================================
Dep. Variable:              Milk in pounds per cow   No. Observations:                  168
Model:             SARIMAX(0, 1, 0)x(1, 1, 1, 12)   Log Likelihood                -534.065
Date:                             Thu, 13 Jul 2017   AIC                           1074.131
Time:                                     00:16:10   BIC                           1083.503
Sample:                                 01-01-1962   HQIC                          1077.934
                                      - 12-01-1975
Covariance Type:                               opg
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
ar.S.L12      -0.0449      0.106     -0.422      0.673      -0.253       0.163
ma.S.L12      -0.5860      0.102     -5.761      0.000      -0.785      -0.387
sigma2        55.5118      5.356     10.365      0.000      45.015      66.009
===================================================================================
Ljung-Box (Q):                       33.48   Jarque-Bera (JB):                32.04
Prob(Q):                              0.76   Prob(JB):                         0.00
Heteroskedasticity (H):               0.69   Skew:                             0.77
Prob(H) (two-sided):                  0.18   Kurtosis:                         4.60
===================================================================================

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

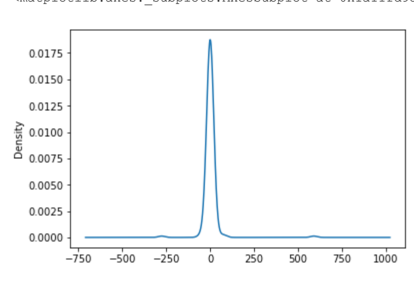
In [41]:
```
results.resid.plot()
```

Out[41]: `<matplotlib.axes._subplots.AxesSubplot at 0x1a11fc47fd0>`



In [42]:
```
results.resid.plot(kind='kde')
```

Out[42]: `<matplotlib.axes._subplots.AxesSubplot at 0x1a11fd9da20>`

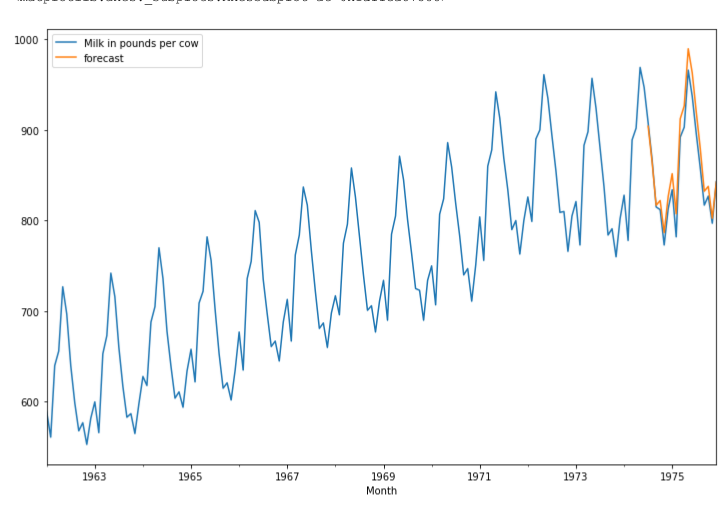


Prediction of Future Values

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

In [43]:
```
df['forecast'] = results.predict(start = 150, end= 168, dynamic= True)
df[['Milk in pounds per cow','forecast']].plot(figsize=(12,8))
```

Out[43]: `<matplotlib.axes._subplots.AxesSubplot at 0x1a11ea87588>`



Forecasting

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

In [39]:
```
df.tail()
```

Out[39]:

Month	Milk in pounds per cow	Milk First Difference	Milk Second Difference	Seasonal Difference	Seasonal First Difference
1975-08-01	858.0	-38.0	3.0	-9.0	3.0
1975-09-01	817.0	-41.0	-3.0	2.0	11.0
1975-10-01	827.0	10.0	51.0	15.0	13.0
1975-11-01	797.0	-30.0	-40.0	24.0	9.0
1975-12-01	843.0	46.0	76.0	30.0	6.0

In [45]:
```
# https://pandas.pydata.org/pandas-docs/stable/timeseries.html
# Alternatives
# pd.date_range(df.index[-1],periods=12,freq='M')
```

In [46]:
```
from pandas.tseries.offsets import DateOffset
```

In [47]:
```
future_dates = [df.index[-1] + DateOffset(months=x) for x in range(0,24) ]
```

In [48]:
```
future_dates
```

Out[48]:
```
[Timestamp('1975-12-01 00:00:00'),
 Timestamp('1976-01-01 00:00:00'),
 Timestamp('1976-02-01 00:00:00'),
 Timestamp('1976-03-01 00:00:00'),
 Timestamp('1976-04-01 00:00:00'),
 Timestamp('1976-05-01 00:00:00'),
 Timestamp('1976-06-01 00:00:00'),
 Timestamp('1976-07-01 00:00:00'),
 Timestamp('1976-08-01 00:00:00'),
 Timestamp('1976-09-01 00:00:00'),
 Timestamp('1976-10-01 00:00:00'),
 Timestamp('1976-11-01 00:00:00'),
 Timestamp('1976-12-01 00:00:00'),
 Timestamp('1977-01-01 00:00:00'),
 Timestamp('1977-02-01 00:00:00'),
 Timestamp('1977-03-01 00:00:00'),
 Timestamp('1977-04-01 00:00:00'),
 Timestamp('1977-05-01 00:00:00'),
 Timestamp('1977-06-01 00:00:00'),
 Timestamp('1977-07-01 00:00:00'),
 Timestamp('1977-08-01 00:00:00'),
 Timestamp('1977-09-01 00:00:00'),
 Timestamp('1977-10-01 00:00:00'),
 Timestamp('1977-11-01 00:00:00')]
```

In [49]:
```
future_dates_df = pd.DataFrame(index=future_dates[1:],columns=df.columns)
```

In [50]:
```
future_df = pd.concat([df,future_dates_df])
```

In [51]:
```
future_df.head()
```

Out[51]:

	Milk in pounds per cow	Milk First Difference	Milk Second Difference	Seasonal Difference	Seasonal First Difference	forecast
1962-01-01	589.0	NaN	NaN	NaN	NaN	NaN
1962-02-01	561.0	-28.0	NaN	NaN	NaN	NaN
1962-03-01	640.0	79.0	107.0	NaN	NaN	NaN
1962-04-01	656.0	16.0	-63.0	NaN	NaN	NaN
1962-05-01	727.0	71.0	55.0	NaN	NaN	NaN

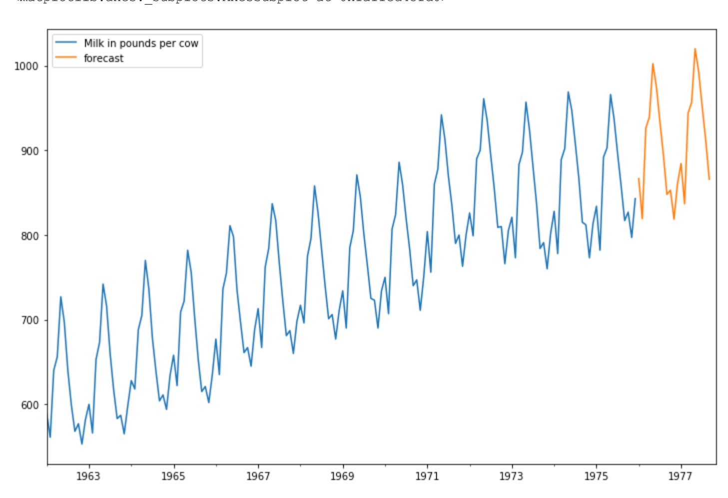
In [52]:
```
future_df.tail()
```

Out[52]:

	Milk in pounds per cow	Milk First Difference	Milk Second Difference	Seasonal Difference	Seasonal First Difference	forecast
1977-07-01	NaN	NaN	NaN	NaN	NaN	NaN
1977-08-01	NaN	NaN	NaN	NaN	NaN	NaN
1977-09-01	NaN	NaN	NaN	NaN	NaN	NaN
1977-10-01	NaN	NaN	NaN	NaN	NaN	NaN
1977-11-01	NaN	NaN	NaN	NaN	NaN	NaN

In [53]:
```
future_df['forecast'] = results.predict(start = 168, end = 188, dynamic= True)
future_df[['Milk in pounds per cow', 'forecast']].plot(figsize=(12, 8))
```

Out[53]: `<matplotlib.axes._subplots.AxesSubplot at 0x1a11ed43fd0>`



Not bad! Pretty cool in fact! I hope this helped you see the potential for ARIMA models, unfortunately a lot of financial data won't follow this sort of behaviour, in fact it will often follow something indicating brownian motion, what is that you ask? Well head on over to the next video section and we'll find out!

Great Job!