Time Series Analysis with Python

In this tutorial, we are going to build a time series model using Python. First, let's import the necessary libraries to perform our little time series analysis.

In [1]:

```
import pandas as pd
import numpy as np
import matplotlib.pylab as plt
%matplotlib inline
from matplotlib.pylab import rcParams
rcParams['figure.figsize'] = 15, 6

from datetime import datetime
```

Loading Time Series Data in Pandas

This tutorial draws from this one and we will be using this dataset which records the daily passenger count for a certain airline company. A cool thing with jupyter notebook is that you can run linux command right here on the notebook simply by preceeding the command with the exclamation sign **!**. For instance, let's look at the first 10 rows of our dataset using the linux command head

In [2]:

```
! head AirPassengers.csv
```

```
Month,#Passengers
1949-01,112
1949-02,118
1949-03,132
1949-04,129
1949-05,121
1949-06,135
1949-07,148
1949-08,148
1949-09,136
```

See that the data is in csv format, with 2 columns. The first column is a string for the month in **%Y-%m** format and the second column is the number of passengers. So here is a useful thing to know when working with time series: **converting strings to datetime**. The python **datetime** utility handles this task using the **strptime** function. See Chris Albon's site for a quick guide. Now back to parsing the data:

In [3]:

```
#define a lambda function that converts a string with format %Y-%m to datetim
e
custom_date_parser = lambda dates: pd.datetime.strptime(dates, '%Y-%m')

#parse the data using pandas read_csv
AirPassengers = pd.read_csv('AirPassengers.csv',
                            parse_dates = ['Month'],
                            index_col   = 'Month',
                            date_parser = custom_date_parser)
AirPassengers.head()
```

Out[3]:

	#Passengers
Month	
1949-01-01	112
1949-02-01	118
1949-03-01	132
1949-04-01	129
1949-05-01	121

In [4]:

```
# Let's see the data types
AirPassengers.dtypes
```

Out[4]:

```
#Passengers    int64
dtype: object
```

Let's walk through the parameters in the read_csv code above

- **parse_dates** tells pandas to parse the specified column as date. values can be boolean or list of ints or names or list of lists or dict, and the default is False.

- **index_col** specifies which column to use as index. When analyzing Time Series in Pandas, it's important that the variable containing the time information be used as the dataFrame index. In the present case, this variable is called **Month**
- **date_parser** This specifies how to parse the date. By default, Pandas usses the format 'YYYY-MM-DD HH:MM:SS'. So if the date we want to parse are not in this format, we need to define a custom parser, just like the **custom_date_parser** above.

We see that the datatype of the Month is now **datetime64[ns]**

In [5]:

```
#Let's convert the dataFrame to a Series, since there is only one column anyw
ays
ts = AirPassengers['#Passengers']
ts.head()
```

Out[5]:

```
Month
1949-01-01    112
1949-02-01    118
1949-03-01    132
1949-04-01    129
1949-05-01    121
Name: #Passengers, dtype: int64
```

Check for Stationarity in Time Series

A Time Series is said to be stationary if it's desciptive statistical properties such **mean**, **variance** remain constant in time. We can assume a series to be stationary if the followings are true (more details here):

- constant mean
- constant variance
- an autocovariance that does not depend on time.

But why should we care about stationarity? Well, most of the statistical methods used in the analysis of time series assume that the series is stationary. In addition, having a stationay series has practical advantage since it would mean that the average behavior of the series at time t will be the same at a time in the future. Hence we can make predictions.

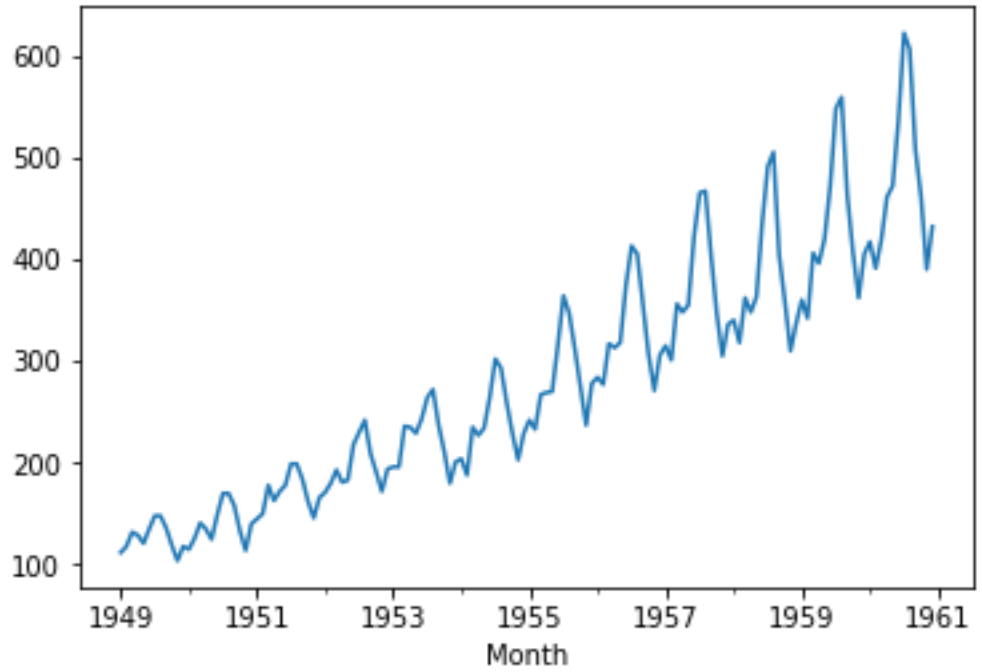
Let's test stationarity in the present data. First we create a simple plot.

In [6]:

```
#visualize the time series with a simple line plot
ts.plot()
```

Out[6]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f2ac3a2d828>
```



We see a clear trend and some seasonality. But it might always be easy to create a visual like this. We can test stationarity using the following methods:

- **Plotting Rolling Statistics:** Plot the moving average or moving variance and see if it varies with time. see pandas documentation for rolling
- **Dickey-Fuller Test:** This is one of the statistical tests for checking stationary. The null hypothesis H_0 is defined as **The Time Series is non-stationary**. i.e. if the **p** value is less than some confidence level α (usually $\alpha=0.05$), we reject the null hypothesis and say that the series is stationary.

see here or thi post

In [22]:

```
from statsmodels.tsa.stattools import adfuller
def test_stationarity(timeseries):

    #Perform Dickey-Fuller test:
    print('Results of Dickey-Fuller Test:')
    dftest = adfuller(timeseries, autolag='AIC')
    dfoutput = pd.Series(dftest[0:4], index=['Test Statistic','p-value','#Lags Used','Number of Observations Used'])
    for key,value in dftest[4].items():
        dfoutput['Critical Value (%s)'%key] = value
    print(dfoutput)
```

In [8]:

```
test_stationarity(ts)
```

```
Results of Dickey-Fuller Test:
Test Statistic                   0.815369
p-value                          0.991880
#Lags Used                      13.000000
Number of Observations Used    130.000000
Critical Value (1%)             -3.481682
Critical Value (5%)             -2.884042
Critical Value (10%)            -2.578770
dtype: float64
```

We may conclude that this is a non-stationary series because:

- The **p-value** is greater than the typical confidence level of 0.05, hence we donot reject the null hypothesis
- The critical values from the Dickey-Fuller Test are way smaller than the Test Statistics

Making a Time Series Stationary

There are a number of reasons why a time series might be non-stationary:

- **Trends**
- **Seasonality**

Hence the basic idea of forecasting with time series data is as follows

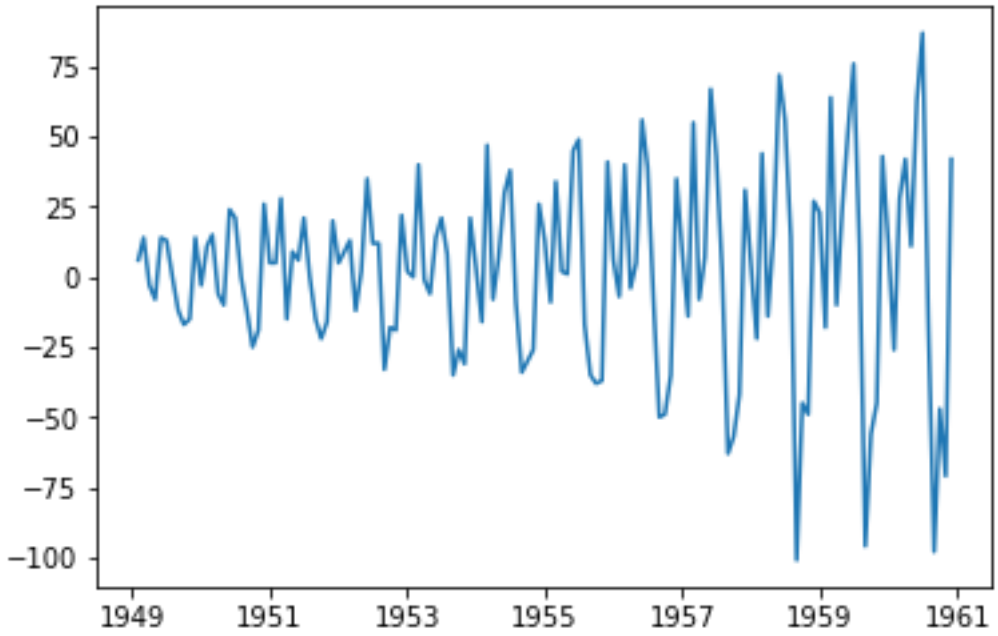
- Estimate trends and seasonalities and remove them to get a "stationary series". This process is know as deseasonalization
- Make predictions using the stationary series. Calculate means etc ..
- Apply the deseasonalization contraints on the predicted value

Differencing

Differencing is a popular method to remove trend and seasonality in a time series. In python, this can be achieved using the **diff()** method of either a series or a dataframe. Check out this post or this one for an overview of performing differencing of time series. More details about other methods of differencing and when to use them can be found here

In [9]:

```
nonseasonal_diff = ts.diff(periods=1)
plt.plot(nonseasonal_diff)
plt.show()
```



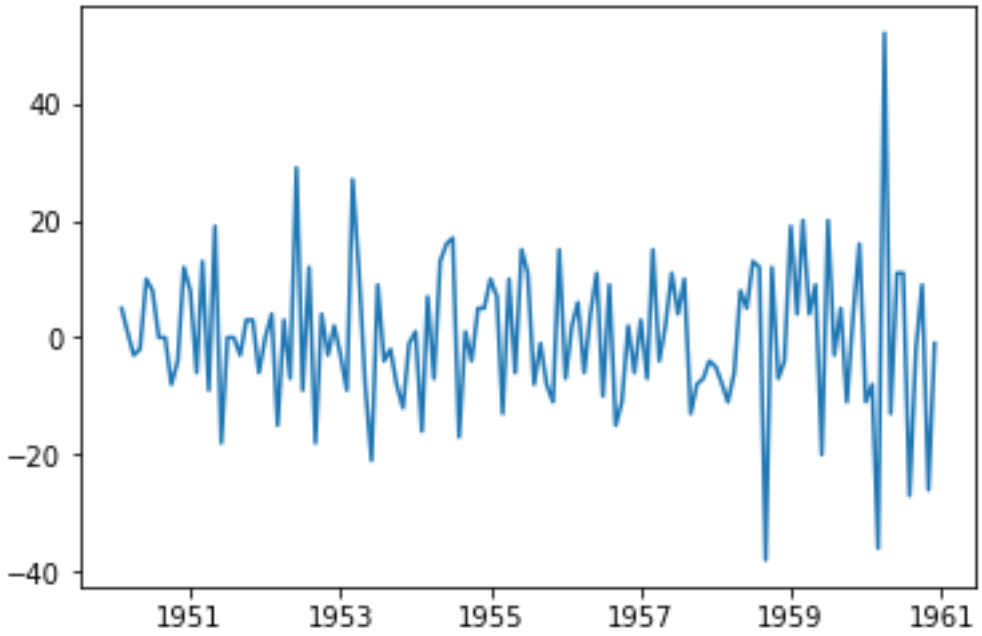
In [10]:

```
test_stationarity(nonseasonal_diff.dropna(inplace=False))
```

```
Results of Dickey-Fuller Test:
Test Statistic                  -2.829267
p-value                          0.054213
#Lags Used                      12.000000
Number of Observations Used    130.000000
Critical Value (1%)             -3.481682
Critical Value (5%)             -2.884042
Critical Value (10%)            -2.578770
dtype: float64
```

In [11]:

```
seasonal_diff = nonseasonal_diff.diff(periods=12)
plt.plot(seasonal_diff)
plt.show()
test_stationarity(seasonal_diff.dropna(inplace=False))
```




```
Results of Dickey-Fuller Test:
Test Statistic                -1.559562e+01
p-value                        1.856512e-28
#Lags Used                     0.000000e+00
Number of Observations Used    1.300000e+02
Critical Value (1%)           -3.481682e+00
Critical Value (5%)           -2.884042e+00
Critical Value (10%)          -2.578770e+00
dtype: float64
```

The Autocorrelation and Partial Autocorrelation plots

These plots help in identifying the order of an Autoregressive ARMA(p, q)

see also pandas visualization documentation for the autocorrelation_plot

In [12]:

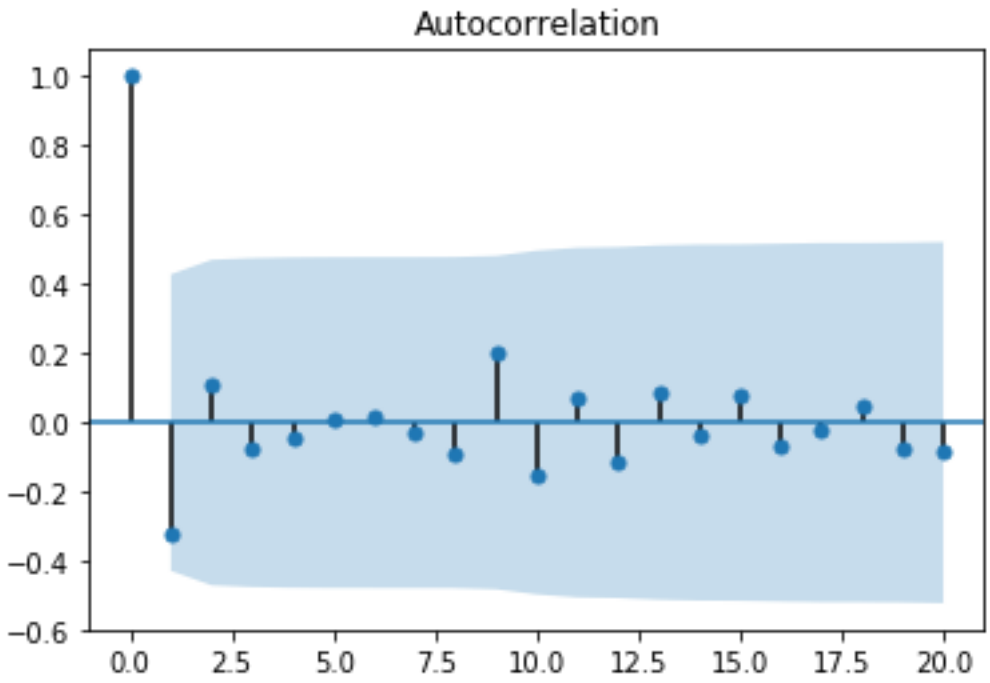
```
#ACF and PACF plots:
from statsmodels.tsa.stattools import acf, pacf
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf
```

In [13]:

```
lag_acf  = acf(seasonal_diff.dropna(inplace=False), nlags=20)
lag_pacf = pacf(seasonal_diff.dropna(inplace=False), nlags=20, method='ols')
```

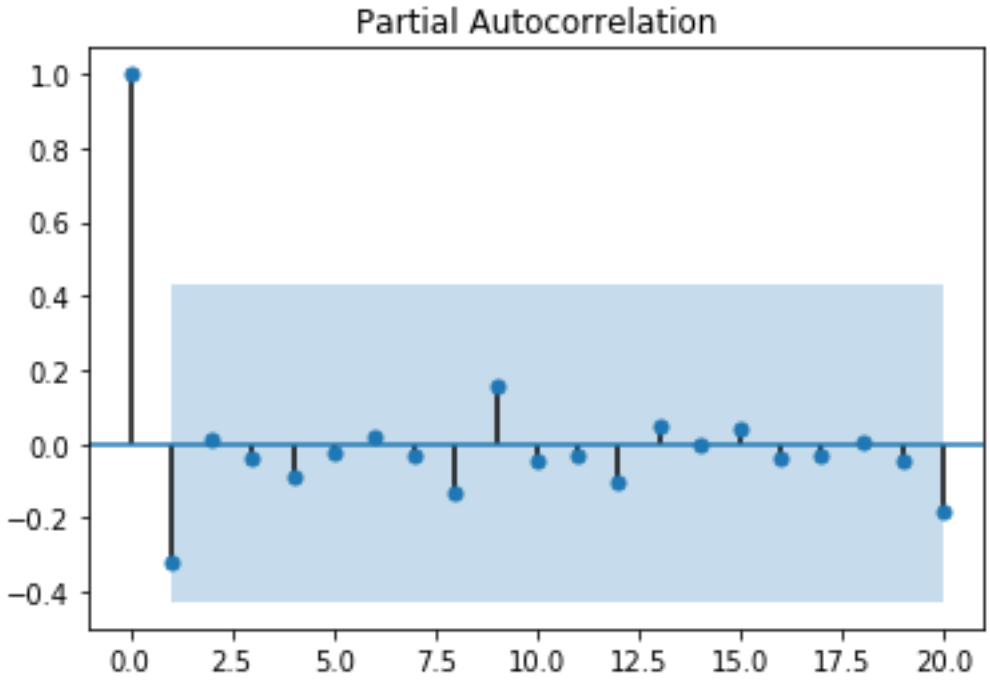
In [14]:

```
p=plot_acf(lag_acf)
```



In [15]:

```
pacf=plot_pacf(lag_acf)
```



In [16]:

```
def PlotAcf(data):
    fig, axes = plt.subplots(nrows=4, figsize=(8, 12))
    fig.tight_layout()

    axes[0].plot(data)
    axes[0].set_title('Raw Data')

    axes[1].acorr(data, maxlags=data.size-1)
    axes[1].set_title('Matplotlib Autocorrelation')

    plot_acf(data, lags=20, ax=axes[2])
    axes[2].set_title('Statsmodels Autocorrelation')

    pd.tools.plotting.autocorrelation_plot(data, ax=axes[3])
    axes[3].set_title('Pandas Autocorrelation')

    # Remove some of the titles and labels that were automatically added
    #for ax in axes.flat:
    #    ax.set(title='', xlabel='')
    #plt.show()
```

In [17]:

```
#PlotAcf(ts_log_diff)
```

Forcasting: ARIMA

In [18]:

```
from statsmodels.tsa.arima_model import ARIMA
from statsmodels.tsa.statespace.sarimax import SARIMAX
```

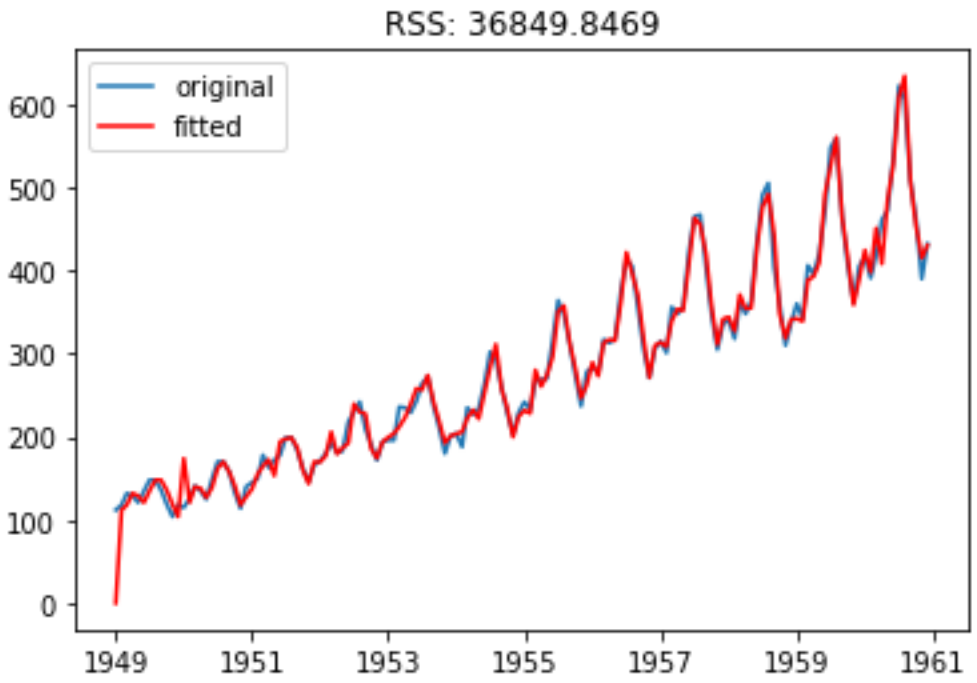
- **Combined AR and MA**

In [19]:

```
model = SARIMAX(ts, trend='n', order=(0,1,0), seasonal_order=(1,1,1,12))
results_ARIMA = model.fit(disp=-1)
plt.plot(ts, label='original')
plt.plot(results_ARIMA.fittedvalues, color='red', label='fitted')
plt.title('RSS: %.4f'% sum((results_ARIMA.fittedvalues-ts)**2))
plt.legend()
```

Out[19]:

```
<matplotlib.legend.Legend at 0x7f2ab1a178d0>
```



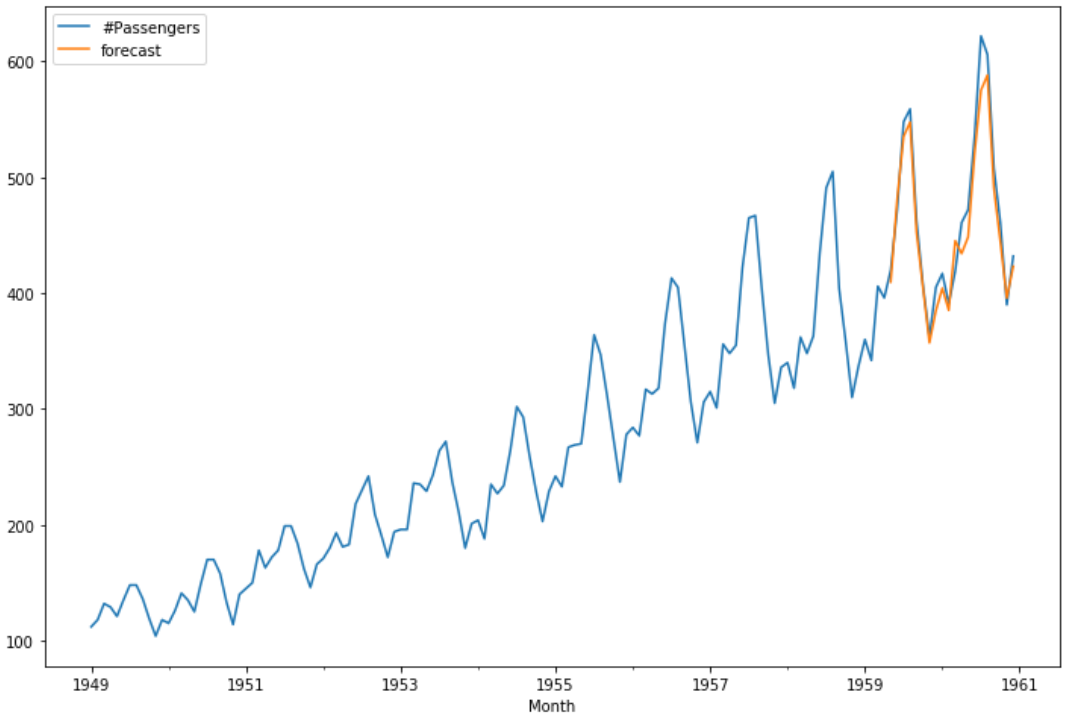
- **Back to original scale**

In [20]:

```
df = pd.DataFrame(ts)
df['forecast'] = results_ARIMA.predict(start = '1959-05-01', end= '1960-12-01
', dynamic= True)
df[['#Passengers', 'forecast']].plot(figsize=(12, 8))
#plt.title('RMSE: %.4f'% np.sqrt(sum((predictions-original_TS)**2)/len(origin
al_TS)))
```

Out[20]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f2ab2827048>
```



In [21]:

```
#predict(results_ARIMA.fittedvalues, ts)
```

Other Tutorials

- Sean Abu's Seaonal ARIMA with Python