Chapter 4

Exploratory Analysis of Time Series

4.1 Overview

seaborn and **matplotlib** are two of the main visualising libraries in Python. Although these libraries are great at plotting univariate or multivariate data, **pandas** has specific visualisation capabilities focussed on time series data. Capabilities which we will explore in this chapter.

4.2 A Simple Plot

In the following examples we will be using data about daily values from the Dow Jones Index for the year 2014.

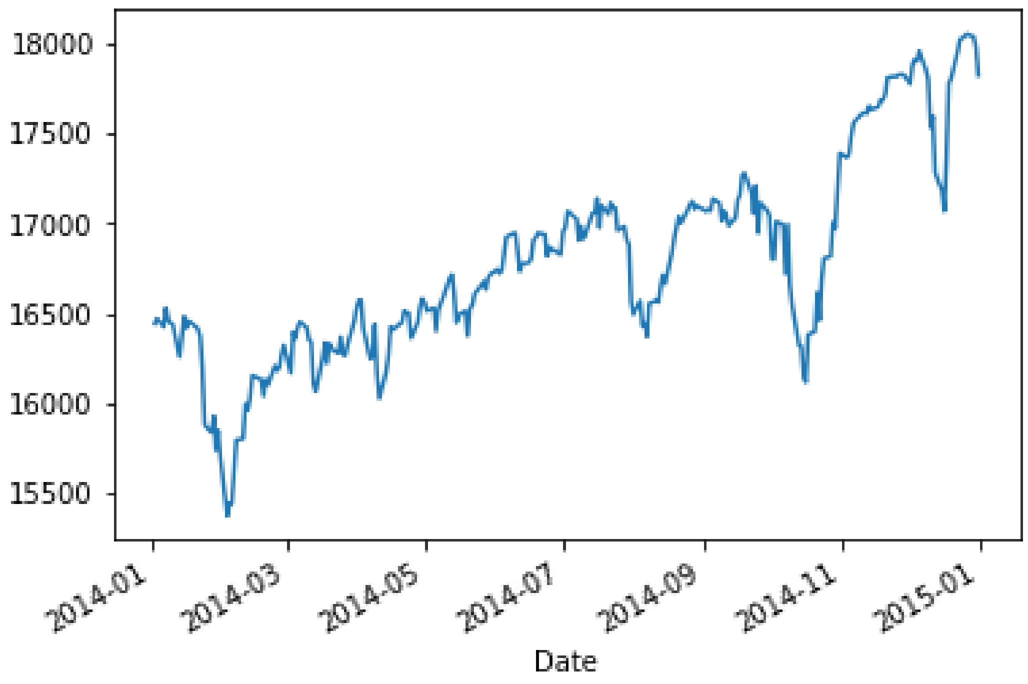
```
>>> import pandas as pd
>>> dji = pd.read_csv("dji.csv", parse_dates=["Date"])
>>> dji = dji.sort_values(by="Date",
ascending=True).set_index("Date")
>>> dji.head()

             DJI.Open   DJI.High    DJI.Low  DJI.Close  DJI.Volume
\
Date
2014-01-02  16572....  16573....  16416....  16441....  809600...
2014-01-03  16456....  16518....  16439....  16469....  727700...
2014-01-06  16474....  16532....  16405....  16425....  893800...
2014-01-07  16429....  16562....  16429....  16530....  812700...
2014-01-08  16527....  16528....  16416....  16462....  103260...

            DJI.Adj.Close
Date
2014-01-02  16441....
2014-01-03  16469....
2014-01-06  16425....
2014-01-07  16530....
2014-01-08  16462....
```

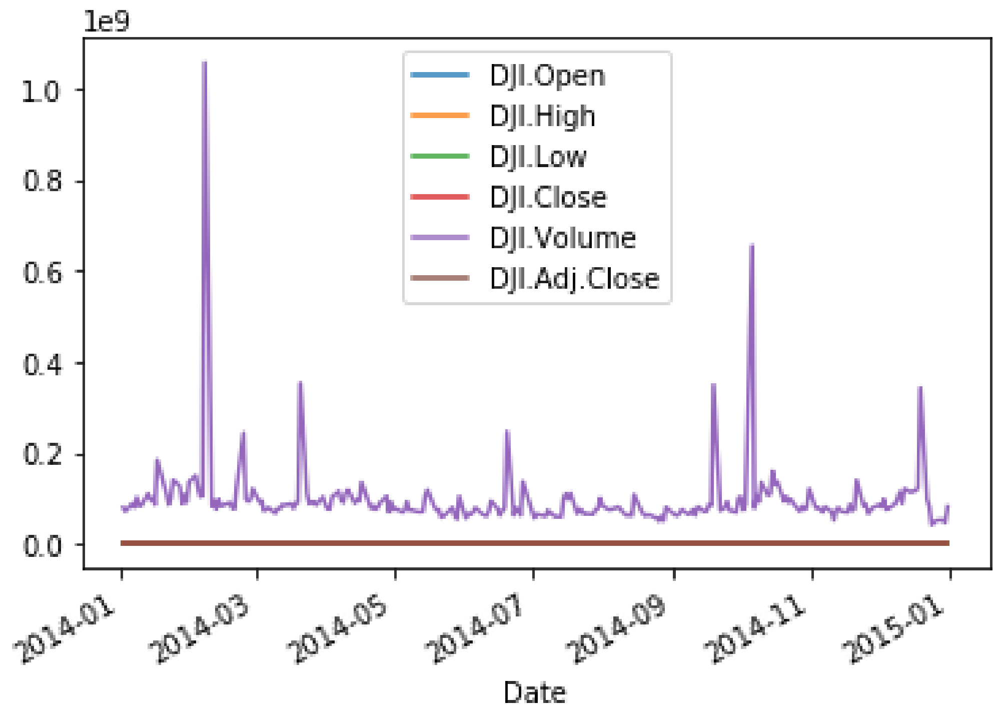
The plotting functionality in **pandas** is built on top of **matplotlib** but in **pandas** plotting is linked to DataFrames. This allows **pandas** to utilize the information that is encapsulated in a DataFrame such as the type of data or the index. For time series data this means **pandas** knows the time dimension needs to be placed on the x-axis and that a line chart is the most suitable default.

```
>>> %matplotlib inline
>>> dji["DJI.Close"].plot();
```



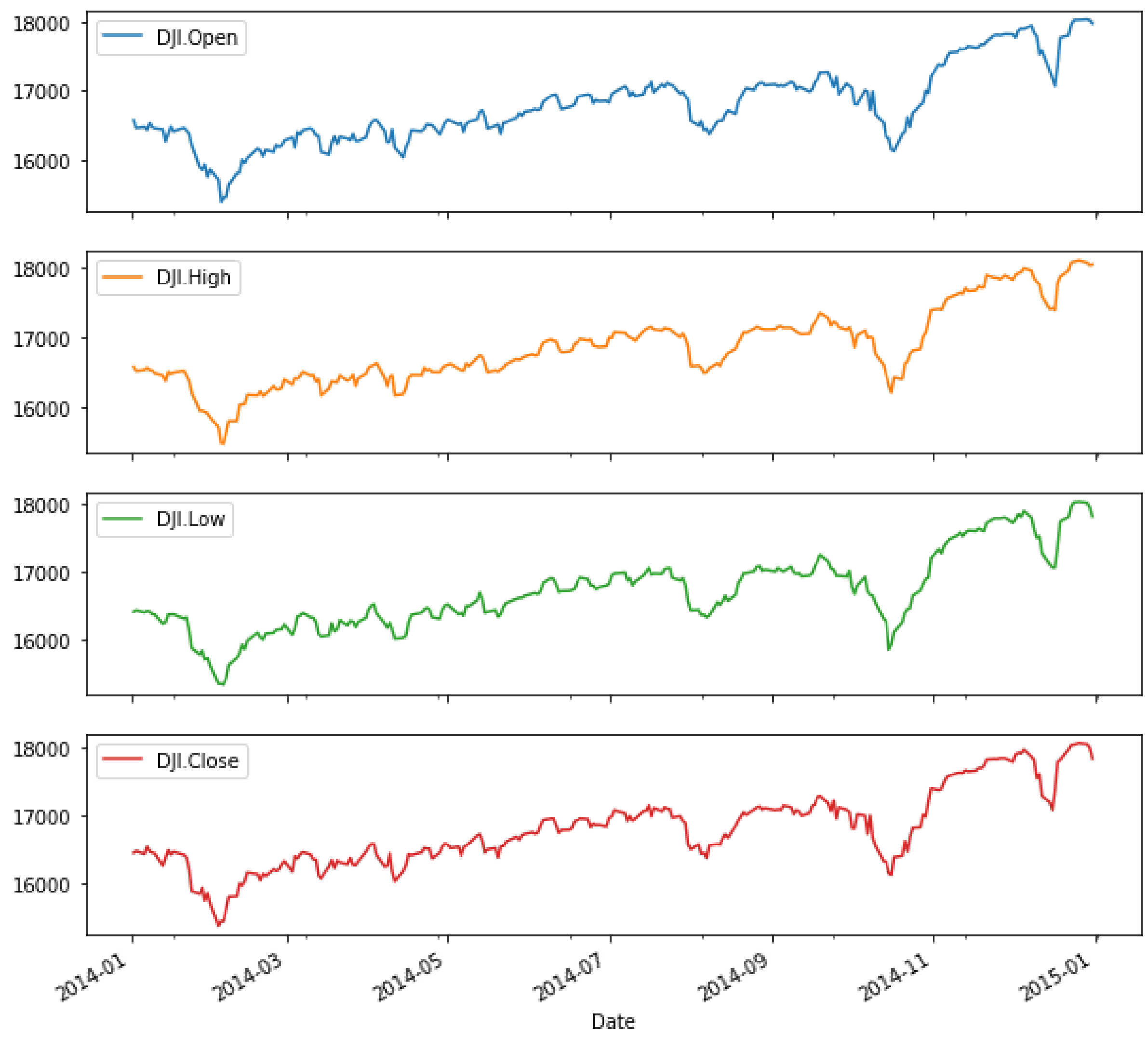
In the above plot we explicitly selected one column. If multiple columns are present **pandas** adds them to the same plot with an accompanying legend.

```
>>> dji.plot();
```



This is convenient for comparing different time series but in this case they're hard to distinguish. We can also decide to not plot them in the same graph using the `subplots` argument.

```
>>> dji[["DJI.Open", "DJI.High", "DJI.Low",
"DJI.Close"]].plot(figsize=(10,10), subplots=True);
```



4.3 Time Series Decomposition

To analyse a time series we typically want to investigate any underlying patterns. These patterns come in various shapes and sizes but broadly speaking we can categorize them into three components: trend, seasonality and residuals.

The trend component identifies the direction in which the time series is moving over a long period of time. With the trend you can answer questions such as: Is the time series increasing or decreasing? Is it doing so at an increasing or decreasing rate?

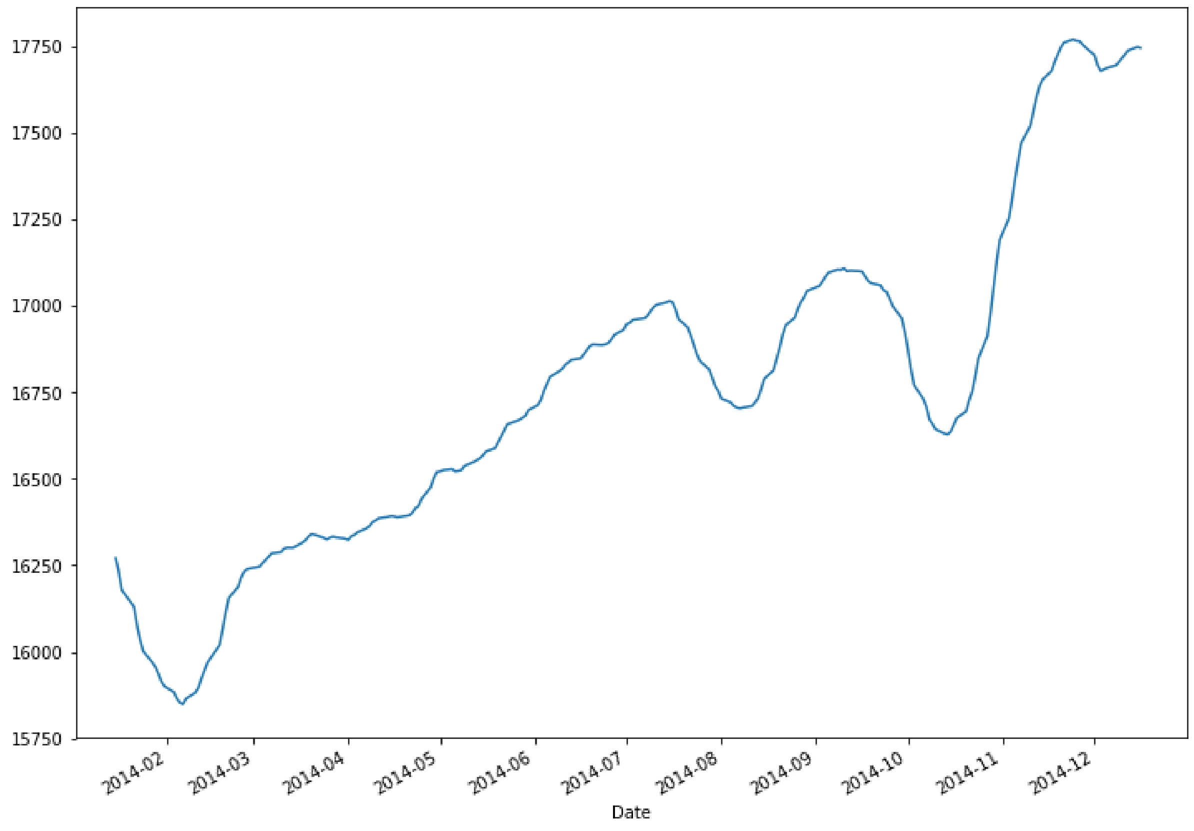
The seasonality component identifies any recurring patterns in the time series. With this component you can answer questions such as: Is the time series higher in the summer than in the winter? Does the time series decrease at the beginning of every week? Since these are recurring patterns they are usually beyond our control. Which is why sometimes the seasonality is removed from the time series and the seasonally adjusted time series is further analysed.

A common first step in time series analysis is to deconstruct the time series into these three components. We will begin by looking at the trend separately and then move on to more sophisticated methods.

4.3.1 Trend

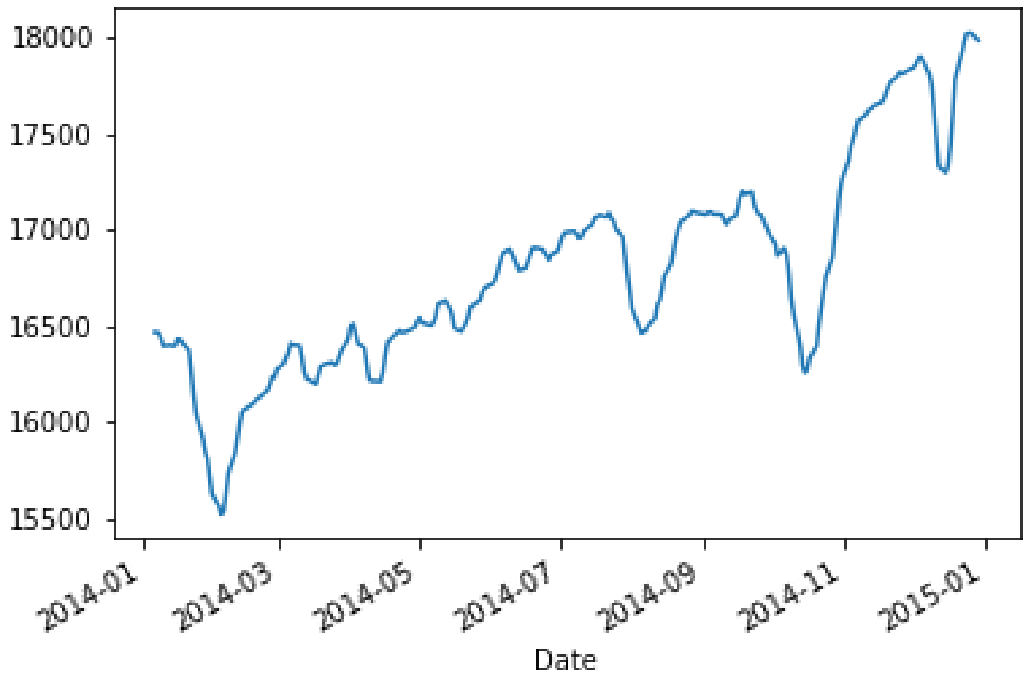
An easy way to identify trend is to take averages over a rolling window. **pandas** offers the convenient `rolling` function to calculate rolling statistics.

```
>>> # calculating 30 day average
>>> dji["DJI.Close"].rolling(window=20, center=True).mean().plot();
```



The size of the `window` is chosen in a way that coincides with the kind of seasonality that you're expecting. In the above case our hypothesis was a monthly seasonality, and we only have data for weekdays, hence the number 20. If instead we assume weekly seasonality we can redraw the graph with a 5 day rolling window (working days only). In either case, the result should be a reasonably smooth line from which you can discern the trend of the time series.

```
>>> dji["DJI.Close"].rolling(window=5, center=True).mean().plot();
```



1. Load in the `barclays.csv` dataset and plot the daily *Close* value (leave the values on the x-axis as is).
2. Calculate a moving average over a suitable period and add it to the plot. Is there a trend present in the time series?

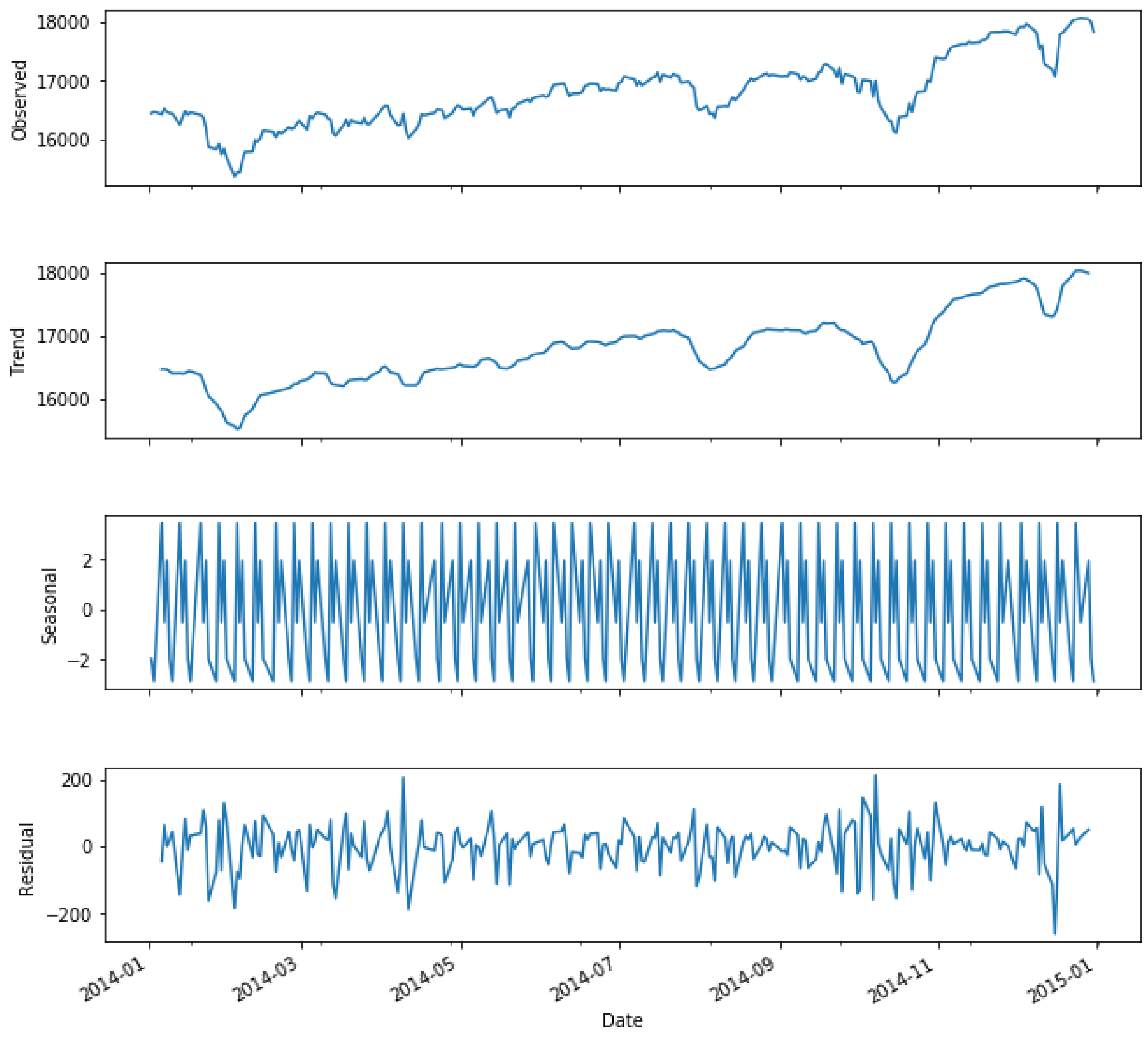
Extension
Find out how to fix the labels on the x-axis such that they don't overlap.

4.3.2 Seasonality

Detecting trend is relatively easy using moving averages. Detecting seasonality however is more complicated and involves simultenously distinguishing between the trend and the residuals components.

Nevertheless there are various methods that allow you to perform seasonal decomposition. We won't go into the details of each method here, that is beyond the scope of the material. Instead we will only show you how to use the implementation of seasonal decomposition in **statsmodels**.

```
>>> from statsmodels.tsa.seasonal import seasonal_decompose
>>> decomposed = seasonal_decompose(dji["DJI.Close"], freq=5)
>>> fig = decomposed.plot()
>>> fig.set_figwidth(10)
>>> fig.set_figheight(10)
```



The trend as estimated by `seasonal_decompose` is in essence just a moving average, hence its similarity to the moving average we created earlier. The seasonal component is estimated by taking the average over the de-trended time series for each season. Then some additional computation is performed to ensure a seasonal value for each point in time.

After deduction of the trend and the seasonal component, what remains are the residual values. We can use these residual values to uncover any patterns not captured by the seasonal or trend component. For example, in the above example we see a sharp decrease at the end of 2014 and an increased variance in October 2014. These findings may help us further down the line when we start building a forecast model.

The above seasonal decomposition is labeled as a “naive” approach. It does not cope well with changes in the seasonal component over time and isn’t robust against outliers. More sophisticated methods exist, such as STL or SEATS (https://en.wikipedia.org/wiki/Seasonal_adjustment), but these are not implemented in **statsmodels**.

4.4 Autocorrelation

Another tool to help us analyse a time series is the autocorrelation. Autocorrelation helps us to identify any relationship between future observations and past observations.

As an example, suppose we had a time series that we just a series of random, normally distributed values, known as white noise in time series analysis. As these values are completely random, there would be no correlation between values in the series that were two time points (lag 2) apart, or three time points apart etc. However, if there is a positive trend then future observations will be *x* times larger than past observations. So there will be correlation between observations that are two, three or more values apart as future values depend on past ones.

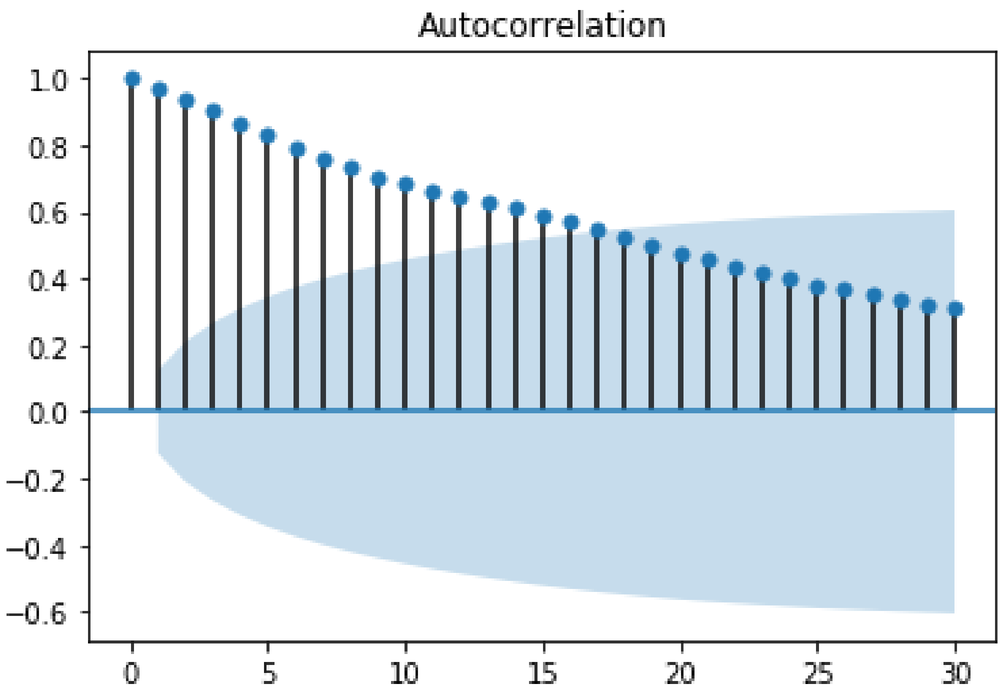
An autocorrelation is the correlation of a variable with lagged values of itself. So for the Dow Jones Index this could mean that we calculate the correlation between each value with the value of the previous day (a lag of 1). If we do this for many lags (i.e. two, three, four etc. days apart) we get a series of autocorrelations which we can analyse visually.

The **statsmodels** package provides the function `acf` to calculate the autocorrelation. Alternatively we can use the autocorrelation plotting function `plot_acf` which includes a 95% confidence interval by default.

```
>>> from statsmodels.tsa.stattools import acf
>>> acf(dji["DJI.Close"], nlags = 30)

array([1.        , 0.97083511, 0.94053691, 0.90487924, 0.86647495,
       0.82946668, 0.79295063, 0.76117492, 0.73114848, 0.70201076,
       0.68182766, 0.66381827, 0.6450658 , 0.62731064, 0.60934462,
       0.58974344, 0.56939623, 0.547497  , 0.52107729, 0.5003139 ,
       0.4774824 , 0.45535916, 0.4351798 , 0.41502874, 0.39770515,
       0.38109563, 0.36702563, 0.35103116, 0.33604838, 0.32213052,
       0.3097104 ])
```

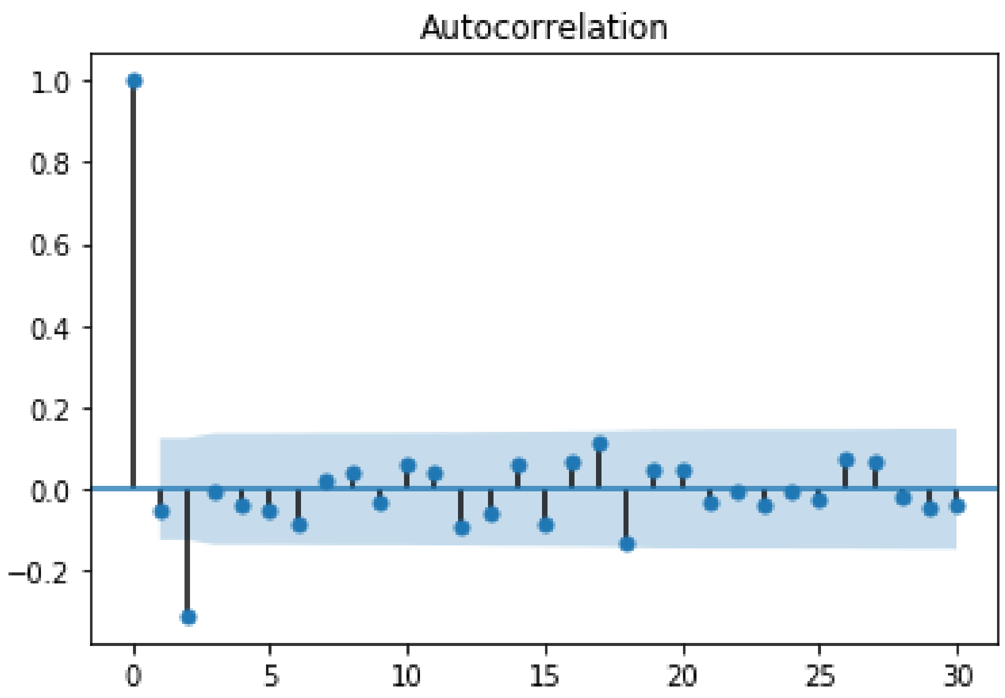
```
>>> from statsmodels.graphics.tsaplots import plot_acf
>>> plot_acf(dji["DJI.Close"], lags = 30);
```



The autocorrelation in itself is not good enough, we must also check if the values are significant, specifically if they are significantly different from zero, which is the value the autocorrelation would take in the case that the series were white noise. Any values outside of the interval are considered significant.

We have applied the autocorrelation function to raw data and we see there are several significant correlation values. That tells us that there is a strong trend component in the time series. Let's apply the same autocorrelation to the residuals from the seasonal decomposition.

```
>>> plot_acf(decomposed.resid.dropna(), lags = 30);
```



The correlation of a time series with itself is always 1 but beyond that there is only one negative correlation that is significant. So, our decomposition has captured almost all of the trend component.

1. Apply seasonal decomposition to the *Close* column from the `barclays.csv` data
2. What does the seasonal component tell you?
3. Is this a good decomposition?
4. Plot the autocorrelation for the raw and detrended data. Does it seem that the decomposition has captured the trend and seasonality?