Chapter 8. Financial Time Series

> *[T]ime is what keeps everything from happening at once.*
>
> —Ray Cummings

Financial time series data is one of the most important types of data in finance. This is data indexed by date and/or time. For example, prices of stocks over time represent financial time series data. Similarly, the EUR/USD exchange rate over time represents a financial time series; the exchange rate is quoted in brief intervals of time, and a collection of such quotes then is a time series of exchange rates.

There is no financial discipline that gets by without considering time an important factor. This mainly is the same as with physics and other sciences. The major tool to cope with time series data in Python is `pandas`. Wes McKinney, the original and main author of `pandas`, started developing the library when working as an analyst at AQR Capital Management, a large hedge fund. It is safe to say that `pandas` has been designed from the ground up to work with financial time series data.

The chapter is mainly based on two financial time series data sets in the form of comma-separated values (CSV) files. It proceeds along the following lines:

"Financial Data"

: This section is about the basics of working with financial times series data using `pandas`: data import, deriving summary statistics, calculating changes over time, and resampling.

"Rolling Statistics"

: In financial analysis, rolling statistics play an important role. These are statistics calculated in general over a fixed time interval that is *rolled forward* over the complete data set. A popular example is simple moving averages. This section illustrates how `pandas` supports the calculation of such statistics.

“Correlation Analysis”

This section presents a case study based on financial time series data for the S&P 500 stock index and the VIX volatility index. It provides some support for the stylized (empirical) fact that both indices are negatively correlated.

“High-Frequency Data”

This section works with high-frequency data, or *tick data*, which has become commonplace in finance. `pandas` again proves powerful in handling such data sets.

Financial Data

This section works with a locally stored financial data set in the form of a CSV file. Technically, such files are simply text files with a data row structure characterized by commas that separate single values. Before importing the data, some package imports and customizations:

```
In [1]: import numpy as np
        import pandas as pd
        from pylab import mpl, plt
        plt.style.use('seaborn')
        mpl.rcParams['font.family'] = 'serif'
        %matplotlib inline
```

Data Import

`pandas` provides a number of different functions and `DataFrame` methods to import data stored in different formats (CSV, SQL, Excel, etc.) and to export data to different formats (see Chapter 9 for more details). The following code uses the `pd.read_csv()` function to import the time series data set from the CSV file:[1]

```
In [2]: filename = '../../source/tr_eikon_eod_data.csv'  ❶

In [3]: f = open(filename, 'r')  ❷
        f.readlines()[:5]  ❷
Out[3]: ['Date,AAPL.O,MSFT.O,INTC.O,AMZN.O,GS.N,SPY,.SPX,.VIX,EUR=,XAU=,GDX,
        ,GLD\n',
```

```
 '2010-01-01,,,,,,,,,1.4323,1096.35,,\n',
 '2010-01-04,30.57282657,30.95,20.88,133.9,173.08,113.33,1132.99,20.04,
,1.4411,1120.0,47.71,109.8\n',
 '2010-01-05,30.625683660000004,30.96,20.87,134.69,176.14,113.63,1136.52,
,19.35,1.4368,1118.65,48.17,109.7\n',
 '2010-01-06,30.138541290000003,30.77,20.8,132.25,174.26,113.71,1137.14,
,19.16,1.4412,1138.5,49.34,111.51\n']

In [4]: data = pd.read_csv(filename,  ❸
                           index_col=0,  ❹
                           parse_dates=True)  ❺

In [5]: data.info()  ❻
        <class 'pandas.core.frame.DataFrame'>
        DatetimeIndex: 2216 entries, 2010-01-01 to 2018-06-29
        Data columns (total 12 columns):
        AAPL.O    2138 non-null float64
        MSFT.O    2138 non-null float64
        INTC.O    2138 non-null float64
        AMZN.O    2138 non-null float64
        GS.N      2138 non-null float64
        SPY       2138 non-null float64
        .SPX      2138 non-null float64
        .VIX      2138 non-null float64
        EUR=      2216 non-null float64
        XAU=      2211 non-null float64
        GDX       2138 non-null float64
        GLD       2138 non-null float64
        dtypes: float64(12)
        memory usage: 225.1 KB
```

❶ Specifies the path and filename.

❷ Shows the first five rows of the raw data (Linux/Mac).

❸ The filename passed to the `pd.read_csv()` function.

❹ Specifies that the first column shall be handled as an index.

❺ Specifies that the index values are of type `datetime`.

❻ The resulting `DataFrame` object.

At this stage, a financial analyst probably takes a first look at the data, either by inspecting or visualizing it (see Figure 8-1):

```
In [6]: data.head()  ❶
Out[6]:
                   AAPL.O  MSFT.O  INTC.O  AMZN.O    GS.N     SPY     .SPX   .VIX  \
    Date
```

```
    2010-01-01        NaN     NaN    NaN     NaN     NaN     NaN      NaN    NaN
    2010-01-04  30.572827  30.950  20.88  133.90  173.08  113.33  1132.99  20.04
    2010-01-05  30.625684  30.960  20.87  134.69  176.14  113.63  1136.52  19.35
    2010-01-06  30.138541  30.770  20.80  132.25  174.26  113.71  1137.14  19.16
    2010-01-07  30.082827  30.452  20.60  130.00  177.67  114.19  1141.69  19.06

                  EUR=     XAU=    GDX     GLD
    Date
    2010-01-01  1.4323  1096.35    NaN     NaN
    2010-01-04  1.4411  1120.00  47.71  109.80
    2010-01-05  1.4368  1118.65  48.17  109.70
    2010-01-06  1.4412  1138.50  49.34  111.51
    2010-01-07  1.4318  1131.90  49.10  110.82

In [7]: data.tail()  ❷
Out[7]:
                AAPL.O  MSFT.O  INTC.O   AMZN.O    GS.N     SPY     .SPX   .VIX  \
    Date
    2018-06-25  182.17   98.39   50.71  1663.15  221.54  271.00  2717.07  17.33
    2018-06-26  184.43   99.08   49.67  1691.09  221.58  271.60  2723.06  15.92
    2018-06-27  184.16   97.54   48.76  1660.51  220.18  269.35  2699.63  17.91
    2018-06-28  185.50   98.63   49.25  1701.45  223.42  270.89  2716.31  16.85
    2018-06-29  185.11   98.61   49.71  1699.80  220.57  271.28  2718.37  16.09

                  EUR=     XAU=    GDX     GLD
    Date
    2018-06-25  1.1702  1265.00  22.01  119.89
    2018-06-26  1.1645  1258.64  21.95  119.26
    2018-06-27  1.1552  1251.62  21.81  118.58
    2018-06-28  1.1567  1247.88  21.93  118.22
    2018-06-29  1.1683  1252.25  22.31  118.65

In [8]: data.plot(figsize=(10, 12), subplots=True);  ❸
```

❶ The first five rows …

❷ … and the final five rows are shown.

❸ This visualizes the complete data set via multiple subplots.

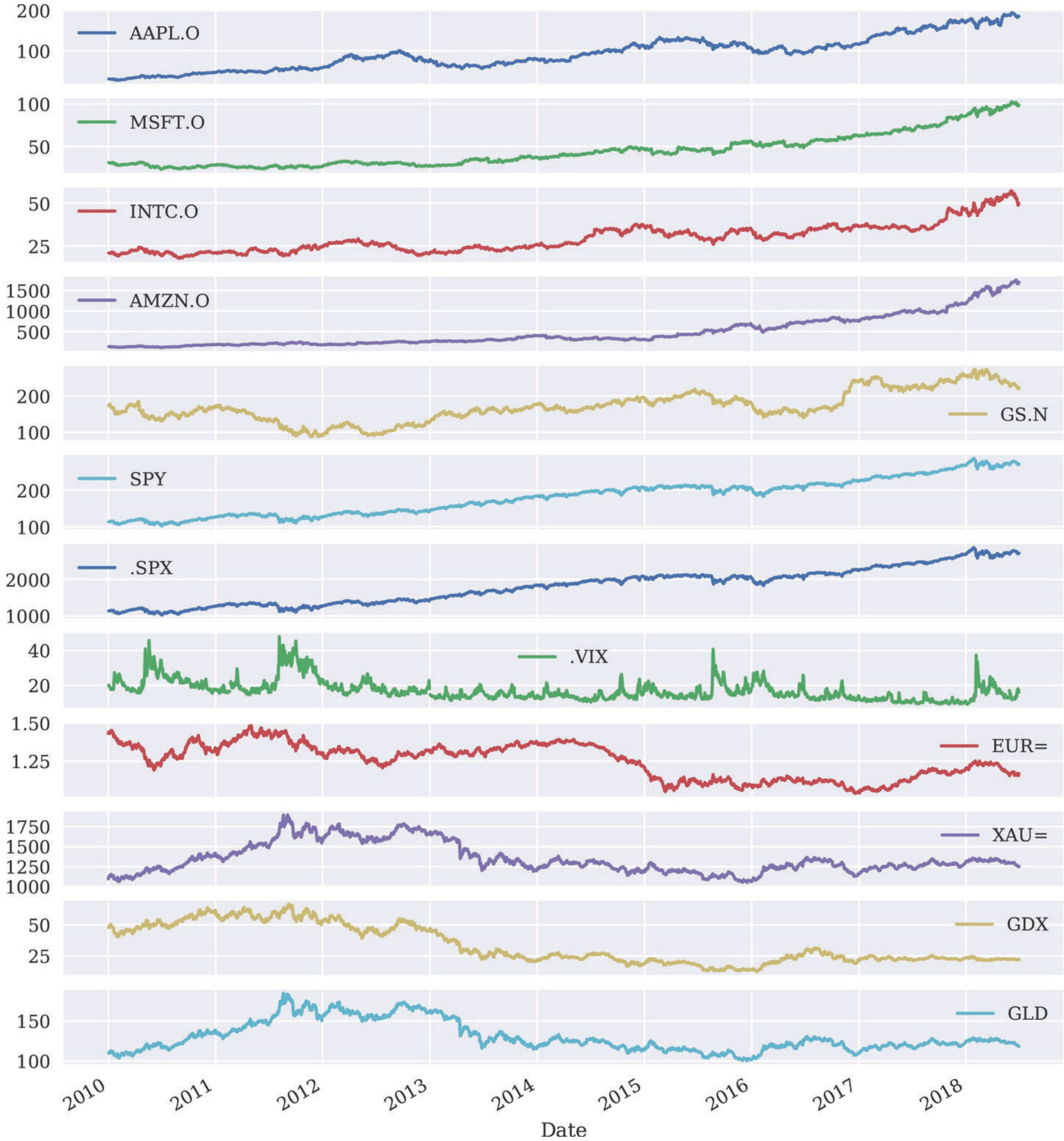

Figure 8-1. Financial time series data as line plots

The data used is from the Thomson Reuters (TR) Eikon Data API. In the TR world symbols for financial instruments are called *Reuters Instrument Codes* (RICs). The financial instruments that the single RICs represent are:

```
In [9]: instruments = ['Apple Stock', 'Microsoft Stock',
                       'Intel Stock', 'Amazon Stock', 'Goldman Sachs Stock',
                       'SPDR S&P 500 ETF Trust', 'S&P 500 Index',
                       'VIX Volatility Index', 'EUR/USD Exchange Rate',
                       'Gold Price', 'VanEck Vectors Gold Miners ETF',
```

```
                              'SPDR Gold Trust']

In [10]: for ric, name in zip(data.columns, instruments):
             print('{:8s} | {}'.format(ric, name))
         AAPL.O   | Apple Stock
         MSFT.O   | Microsoft Stock
         INTC.O   | Intel Stock
         AMZN.O   | Amazon Stock
         GS.N     | Goldman Sachs Stock
         SPY      | SPDR S&P 500 ETF Trust
         .SPX     | S&P 500 Index
         .VIX     | VIX Volatility Index
         EUR=     | EUR/USD Exchange Rate
         XAU=     | Gold Price
         GDX      | VanEck Vectors Gold Miners ETF
         GLD      | SPDR Gold Trust
```

Summary Statistics

The next step the financial analyst might take is to have a look at different summary statistics for the data set to get a “feeling” for what it is all about:

```
In [11]: data.info()  ❶
         <class 'pandas.core.frame.DataFrame'>
         DatetimeIndex: 2216 entries, 2010-01-01 to 2018-06-29
         Data columns (total 12 columns):
         AAPL.O    2138 non-null float64
         MSFT.O    2138 non-null float64
         INTC.O    2138 non-null float64
         AMZN.O    2138 non-null float64
         GS.N      2138 non-null float64
         SPY       2138 non-null float64
         .SPX      2138 non-null float64
         .VIX      2138 non-null float64
         EUR=      2216 non-null float64
         XAU=      2211 non-null float64
         GDX       2138 non-null float64
         GLD       2138 non-null float64
         dtypes: float64(12)
         memory usage: 225.1 KB

In [12]: data.describe().round(2)  ❷
Out[12]:
           AAPL.O   MSFT.O   INTC.O   AMZN.O     GS.N      SPY     .SPX     .VIX  \
    count  2138.00  2138.00  2138.00  2138.00  2138.00  2138.00  2138.00  2138.00
    mean     93.46    44.56    29.36   480.46   170.22   180.32  1802.71    17.03
    std      40.55    19.53     8.17   372.31    42.48    48.19   483.34     5.88
    min      27.44    23.01    17.66   108.61    87.70   102.20  1022.58     9.14
```

```
25%      60.29    28.57    22.51    213.60   146.61   133.99  1338.57    13.07
50%      90.55    39.66    27.33    322.06   164.43   186.32  1863.08    15.58
75%     117.24    54.37    34.71    698.85   192.13   210.99  2108.94    19.07
max     193.98   102.49    57.08   1750.08   273.38   286.58  2872.87    48.00

           EUR=     XAU=      GDX      GLD
count   2216.00  2211.00  2138.00  2138.00
mean       1.25  1349.01    33.57   130.09
std        0.11   188.75    15.17    18.78
min        1.04  1051.36    12.47   100.50
25%        1.13  1221.53    22.14   117.40
50%        1.27  1292.61    25.62   124.00
75%        1.35  1428.24    48.34   139.00
max        1.48  1898.99    66.63   184.59
```

❶ `info()` gives some metainformation about the `DataFrame` object.

❷ `describe()` provides useful standard statistics per column.

> **QUICK INSIGHTS**
>
> `pandas` provides a number of methods to gain a quick overview over newly imported financial time series data sets, such as `info()` and `describe()`. They also allow for quick checks of whether the importing procedure worked as desired (e.g., whether the `DataFrame` object indeed has an index of type `DatetimeIndex`).

There are also options, of course, to customize what types of statistic to derive and display:

```
In [13]: data.mean()  ❶
Out[13]: AAPL.O       93.455973
         MSFT.O       44.561115
         INTC.O       29.364192
         AMZN.O      480.461251
         GS.N        170.216221
         SPY         180.323029
         .SPX       1802.713106
         .VIX         17.027133
         EUR=          1.248587
         XAU=       1349.014130
         GDX          33.566525
         GLD         130.086590
         dtype: float64

In [14]: data.aggregate([min,  ❷
```

```
                                np.mean,   ❸
                                np.std,   ❹
                                np.median,   ❺
                                max]   ❻
                ).round(2)
Out[14]:
               AAPL.O  MSFT.O  INTC.O   AMZN.O    GS.N     SPY     .SPX   .VIX  EUR=  \
    min         27.44   23.01   17.66   108.61   87.70  102.20  1022.58   9.14  1.04
    mean        93.46   44.56   29.36   480.46  170.22  180.32  1802.71  17.03  1.25
    std         40.55   19.53    8.17   372.31   42.48   48.19   483.34   5.88  0.11
    median      90.55   39.66   27.33   322.06  164.43  186.32  1863.08  15.58  1.27
    max        193.98  102.49   57.08  1750.08  273.38  286.58  2872.87  48.00  1.48

                  XAU=    GDX     GLD
    min        1051.36  12.47  100.50
    mean       1349.01  33.57  130.09
    std         188.75  15.17   18.78
    median     1292.61  25.62  124.00
    max        1898.99  66.63  184.59
```

❶ The mean value per column.
❷ The minimum value per column.
❸ The mean value per column.
❹ The standard deviation per column.
❺ The median per column.
❻ The maximum value per column.

Using the `aggregate()` method also allows one to pass custom functions.

Changes over Time

Statistical analysis methods are often based on changes over time and not the absolute values themselves. There are multiple options to calculate the changes in a time series over time, including absolute differences, percentage changes, and logarithmic (log) returns.

First, the absolute differences, for which `pandas` provides a special method:

```
In [15]: data.diff().head()   ❶
Out[15]:
                     AAPL.O  MSFT.O  INTC.O  AMZN.O  GS.N  SPY  .SPX  .VIX    EUR=  \
    Date
    2010-01-01          NaN     NaN     NaN     NaN   NaN  NaN   NaN   NaN     NaN
    2010-01-04          NaN     NaN     NaN     NaN   NaN  NaN   NaN   NaN  0.0088
```

```
2010-01-05  0.052857   0.010    -0.01    0.79  3.06  0.30  3.53 -0.69 -0.0043
2010-01-06 -0.487142  -0.190    -0.07   -2.44 -1.88  0.08  0.62 -0.19  0.0044
2010-01-07 -0.055714  -0.318    -0.20   -2.25  3.41  0.48  4.55 -0.10 -0.0094

             XAU=   GDX   GLD
Date
2010-01-01    NaN   NaN   NaN
2010-01-04  23.65   NaN   NaN
2010-01-05  -1.35  0.46 -0.10
2010-01-06  19.85  1.17  1.81
2010-01-07  -6.60 -0.24 -0.69
```

```
In [16]: data.diff().mean()  ❷
Out[16]: AAPL.O    0.064737
         MSFT.O    0.031246
         INTC.O    0.013540
         AMZN.O    0.706608
         GS.N      0.028224
         SPY       0.072103
         .SPX      0.732659
         .VIX     -0.019583
         EUR=     -0.000119
         XAU=      0.041887
         GDX      -0.015071
         GLD      -0.003455
         dtype: float64
```

❶ `diff()` provides the absolute changes between two index values.

❷ Of course, aggregation operations can be applied in addition.

From a statistics point of view, absolute changes are not optimal because they are dependent on the scale of the time series data itself. Therefore, percentage changes are usually preferred. The following code derives the percentage changes or percentage returns (also: simple returns) in a financial context and visualizes their mean values per column (see Figure 8-2):

```
In [17]: data.pct_change().round(3).head()  ❶
Out[17]:
             AAPL.O  MSFT.O  INTC.O  AMZN.O   GS.N    SPY   .SPX   .VIX   EUR=  \
Date
2010-01-01      NaN     NaN     NaN     NaN    NaN    NaN    NaN    NaN    NaN
2010-01-04      NaN     NaN     NaN     NaN    NaN    NaN    NaN    NaN  0.006
2010-01-05    0.002   0.000  -0.000   0.006  0.018  0.003  0.003 -0.034 -0.003
2010-01-06   -0.016  -0.006  -0.003  -0.018 -0.011  0.001  0.001 -0.010  0.003
2010-01-07   -0.002  -0.010  -0.010  -0.017  0.020  0.004  0.004 -0.005 -0.007
```

```
              XAU=    GDX    GLD
Date
2010-01-01     NaN    NaN    NaN
2010-01-04   0.022    NaN    NaN
2010-01-05  -0.001  0.010 -0.001
2010-01-06   0.018  0.024  0.016
2010-01-07  -0.006 -0.005 -0.006

In [18]: data.pct_change().mean().plot(kind='bar', figsize=(10, 6));  ❷
```

❶ `pct_change()` calculates the percentage change between two index values.

❷ The mean values of the results are visualized as a bar plot.

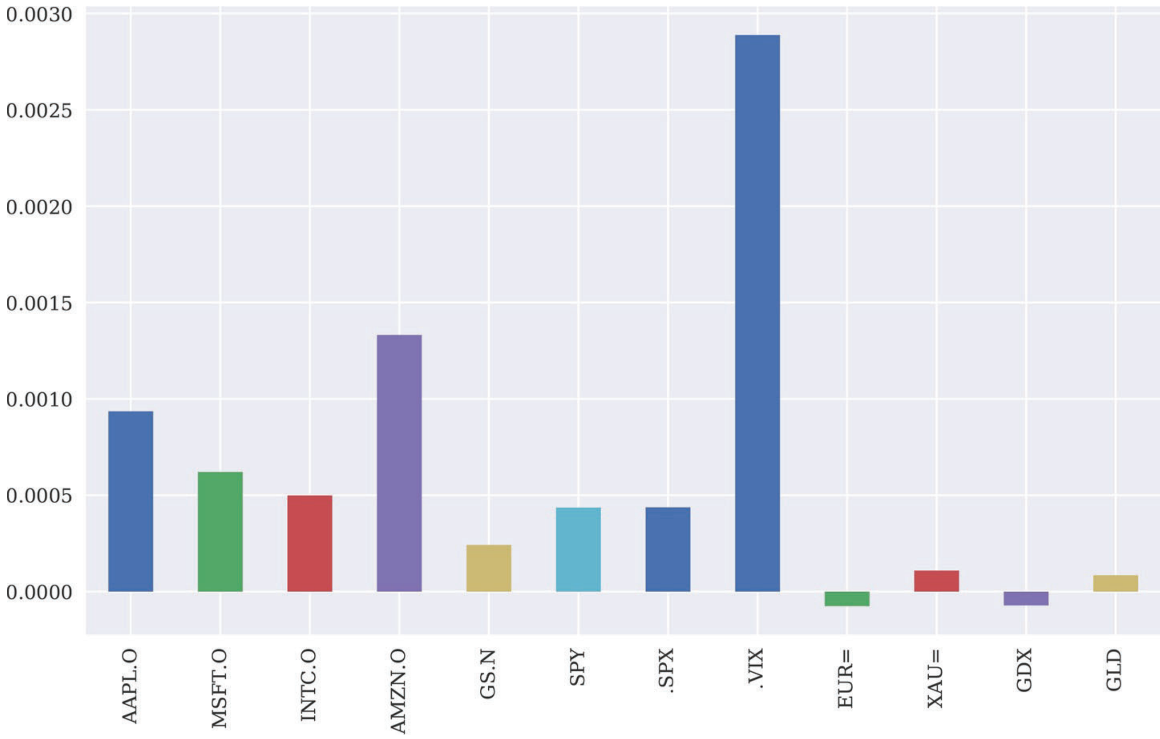

Figure 8-2. Mean values of percentage changes as bar plot

As an alternative to percentage returns, log returns can be used. In some scenarios, they are easier to handle and therefore often preferred in a financial context.[2] Figure 8-3 shows the cumulative log returns for the single financial time series. This type of plot leads to some form of *normalization*:

```
In [19]: rets = np.log(data / data.shift(1))  ❶

In [20]: rets.head().round(3)  ❷
Out[20]:
```

```
            AAPL.O  MSFT.O  INTC.O  AMZN.O    GS.N    SPY    .SPX    .VIX    EUR=  \
Date
2010-01-01     NaN     NaN     NaN     NaN     NaN    NaN     NaN     NaN     NaN
2010-01-04     NaN     NaN     NaN     NaN     NaN    NaN     NaN     NaN   0.006
2010-01-05   0.002   0.000  -0.000   0.006   0.018  0.003   0.003  -0.035  -0.003
2010-01-06  -0.016  -0.006  -0.003  -0.018  -0.011  0.001   0.001  -0.010   0.003
2010-01-07  -0.002  -0.010  -0.010  -0.017   0.019  0.004   0.004  -0.005  -0.007

             XAU=    GDX    GLD
Date
2010-01-01    NaN    NaN    NaN
2010-01-04  0.021    NaN    NaN
2010-01-05 -0.001  0.010 -0.001
2010-01-06  0.018  0.024  0.016
2010-01-07 -0.006 -0.005 -0.006

In [21]: rets.cumsum().apply(np.exp).plot(figsize=(10, 6));  ❸
```

❶ Calculates the log returns in vectorized fashion.

❷ A subset of the results.

❸ Plots the cumulative log returns over time; first the `cumsum()` method is called, then `np.exp()` is applied to the results.

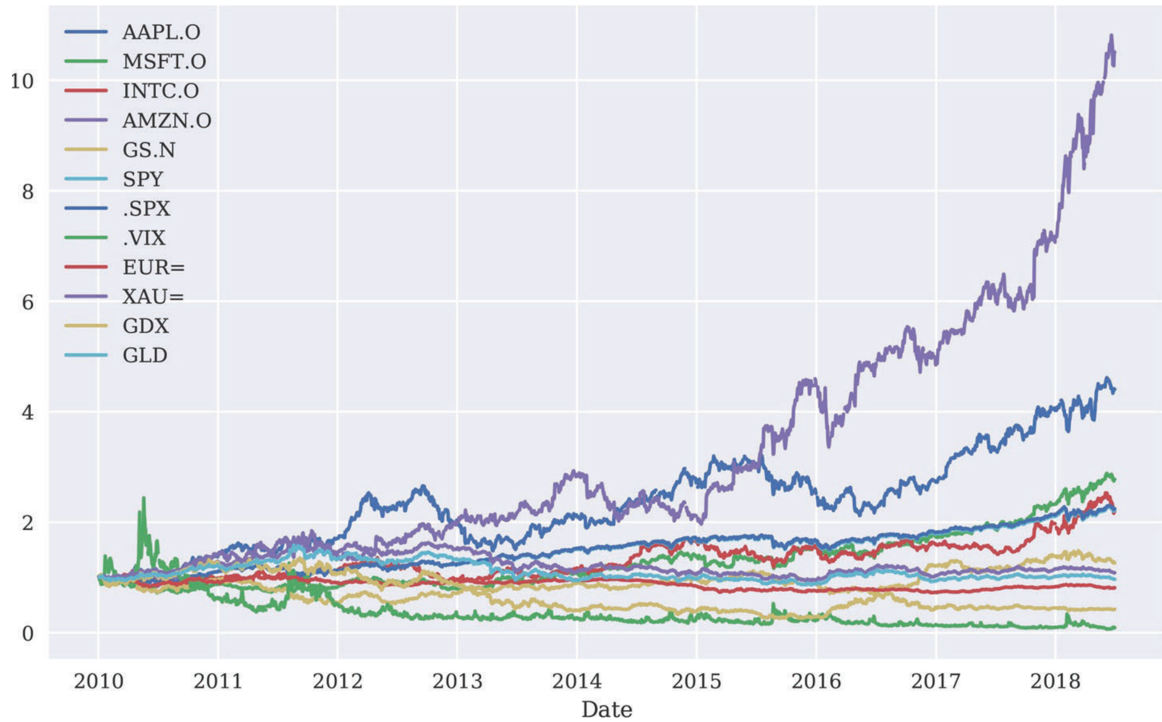

Figure 8-3. Cumulative log returns over time

Resampling

Resampling is an important operation on financial time series data. Usually this takes the form of *downsampling*, meaning that, for example, a tick data series is resampled to one-minute intervals or a time series with daily observations is resampled to one with weekly or monthly observations (as shown in Figure 8-4):

```
In [22]: data.resample('1w', label='right').last().head()  ❶
Out[22]:
                 AAPL.O  MSFT.O  INTC.O  AMZN.O   GS.N     SPY     .SPX   .VIX  \
    Date
    2010-01-03         NaN     NaN     NaN     NaN     NaN     NaN      NaN    NaN
    2010-01-10   30.282827   30.66   20.83  133.52  174.31  114.57  1144.98  18.13
    2010-01-17   29.418542   30.86   20.80  127.14  165.21  113.64  1136.03  17.91
    2010-01-24   28.249972   28.96   19.91  121.43  154.12  109.21  1091.76  27.31
    2010-01-31   27.437544   28.18   19.40  125.41  148.72  107.39  1073.87  24.62

                 EUR=     XAU=    GDX     GLD
    Date
    2010-01-03  1.4323  1096.35    NaN     NaN
    2010-01-10  1.4412  1136.10  49.84  111.37
    2010-01-17  1.4382  1129.90  47.42  110.86
    2010-01-24  1.4137  1092.60  43.79  107.17
    2010-01-31  1.3862  1081.05  40.72  105.96

In [23]: data.resample('1m', label='right').last().head()  ❷
Out[23]:
                 AAPL.O   MSFT.O  INTC.O  AMZN.O    GS.N       SPY     .SPX  \
    Date
    2010-01-31  27.437544  28.1800   19.40  125.41  148.72  107.3900  1073.87
    2010-02-28  29.231399  28.6700   20.53  118.40  156.35  110.7400  1104.49
    2010-03-31  33.571395  29.2875   22.29  135.77  170.63  117.0000  1169.43
    2010-04-30  37.298534  30.5350   22.84  137.10  145.20  118.8125  1186.69
    2010-05-31  36.697106  25.8000   21.42  125.46  144.26  109.3690  1089.41

                .VIX    EUR=     XAU=    GDX      GLD
    Date
    2010-01-31  24.62  1.3862  1081.05  40.72  105.960
    2010-02-28  19.50  1.3625  1116.10  43.89  109.430
    2010-03-31  17.59  1.3510  1112.80  44.41  108.950
    2010-04-30  22.05  1.3295  1178.25  50.51  115.360
    2010-05-31  32.07  1.2305  1215.71  49.86  118.881

In [24]: rets.cumsum().apply(np.exp). resample('1m', label='right').last(
                                  ).plot(figsize=(10, 6));  ❸
```

❶ EOD data gets resampled to *weekly* time intervals …

❷ … and *monthly* time intervals.

❸ This plots the cumulative log returns over time: first, the `cumsum()` method is called, then `np.exp()` is applied to the results; finally, the resampling takes place.

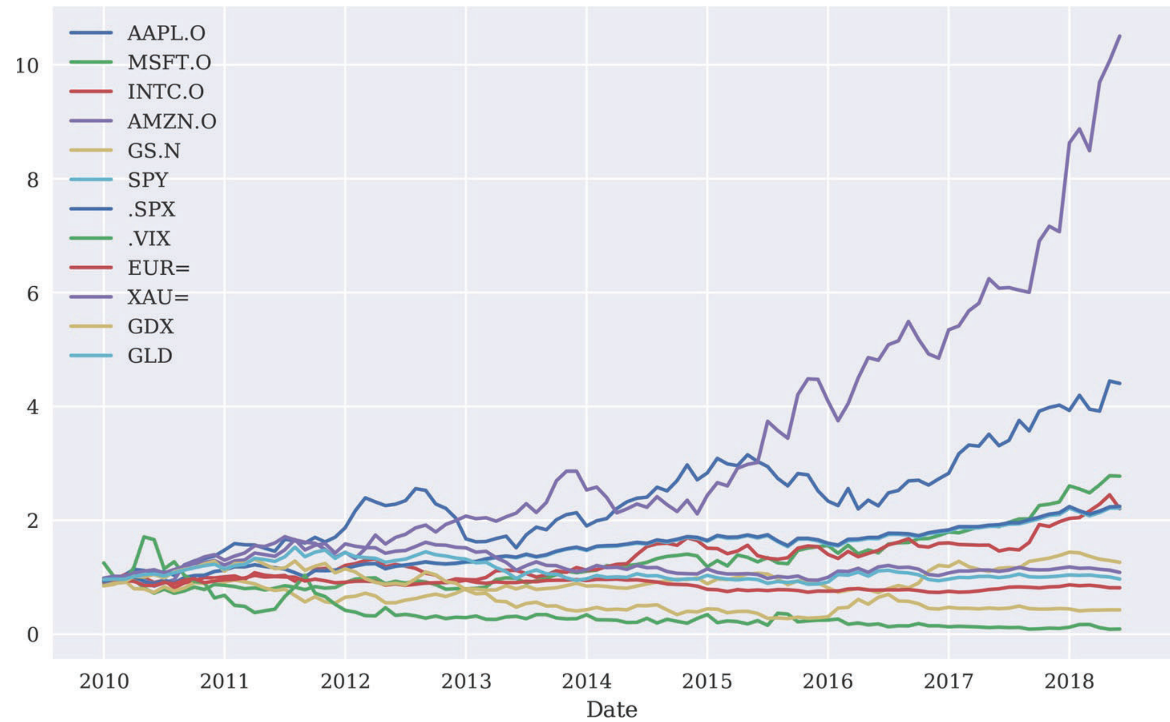

Figure 8-4. Resampled cumulative log returns over time (monthly)

> **AVOIDING FORESIGHT BIAS**
>
> When resampling, `pandas` takes by default in many cases the left label (or index value) of the interval. To be financially consistent, make sure to use the right label (index value) and in general the last available data point in the interval. Otherwise, a foresight bias might sneak into the financial analysis.[3]

Rolling Statistics

It is financial tradition to work with *rolling statistics*, often also called *financial indicators* or *financial studies*. Such rolling statistics are basic tools for financial chartists and technical traders, for example. This section works with a single financial time series only:

```
In [25]: sym = 'AAPL.O'

In [26]: data = pd.DataFrame(data[sym]).dropna()

In [27]: data.tail()
Out[27]:             AAPL.O
         Date
         2018-06-25  182.17
         2018-06-26  184.43
         2018-06-27  184.16
         2018-06-28  185.50
         2018-06-29  185.11
```

An Overview

It is straightforward to derive standard rolling statistics with `pandas`:

```
In [28]: window = 20  ❶

In [29]: data['min'] = data[sym].rolling(window=window).min()  ❷

In [30]: data['mean'] = data[sym].rolling(window=window).mean()  ❸

In [31]: data['std'] = data[sym].rolling(window=window).std()  ❹

In [32]: data['median'] = data[sym].rolling(window=window).median()  ❺

In [33]: data['max'] = data[sym].rolling(window=window).max()  ❻

In [34]: data['ewma'] = data[sym].ewm(halflife=0.5, min_periods=window).mean()  ❼
```

❶ Defines the window; i.e., the number of index values to include.
❷ Calculates the rolling minimum value.
❸ Calculates the rolling mean value.
❹ Calculates the rolling standard deviation.
❺ Calculates the rolling median value.
❻ Calculates the rolling maximum value.
❼ Calculates the exponentially weighted moving average, with decay in terms of a half life of `0.5`.

To derive more specialized financial indicators, additional packages are generally needed (see, for instance, the financial plots with `Cufflinks` in "Interactive 2D Plotting"). Custom ones can also easily be applied via the `apply()` method.

The following code shows a subset of the results and visualizes a selection of the calculated rolling statistics (see Figure 8-5):

```
In [35]: data.dropna().head()
Out[35]:
                AAPL.O        min       mean       std     median        max  \
    Date
    2010-02-01  27.818544  27.437544  29.580892  0.933650  29.821542  30.719969
    2010-02-02  27.979972  27.437544  29.451249  0.968048  29.711113  30.719969
    2010-02-03  28.461400  27.437544  29.343035  0.950665  29.685970  30.719969
    2010-02-04  27.435687  27.435687  29.207892  1.021129  29.547113  30.719969
    2010-02-05  27.922829  27.435687  29.099892  1.037811  29.419256  30.719969

                    ewma
    Date
    2010-02-01  27.805432
    2010-02-02  27.936337
    2010-02-03  28.330134
    2010-02-04  27.659299
    2010-02-05  27.856947

In [36]: ax = data[['min', 'mean', 'max']].iloc[-200:].plot(
             figsize=(10, 6), style=['g--', 'r--', 'g--'], lw=0.8)  ❶
         data[sym].iloc[-200:].plot(ax=ax, lw=2.0);  ❷
```

❶ Plots three rolling statistics for the final 200 data rows.

❷ Adds the original time series data to the plot.

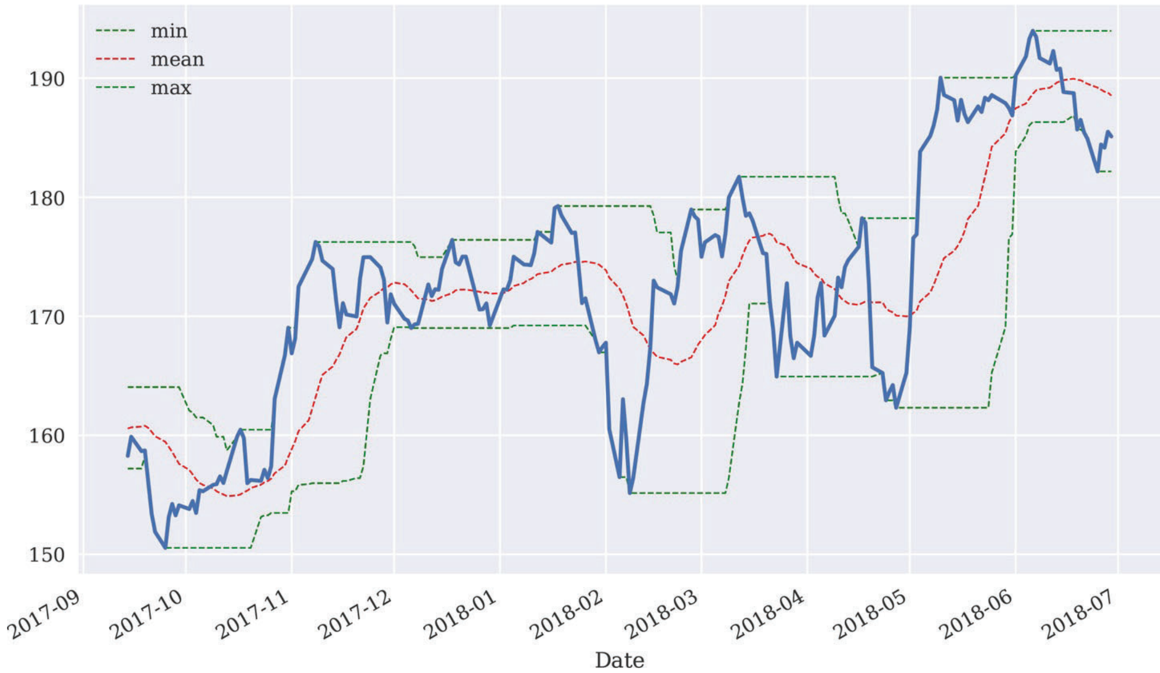

Figure 8-5. Rolling statistics for minimum, mean, maximum values

A Technical Analysis Example

Rolling statistics are a major tool in the so-called *technical analysis* of stocks, as compared to the fundamental analysis which focuses, for instance, on financial reports and the strategic positions of the company whose stock is being analyzed.

A decades-old trading strategy based on technical analysis is using two *simple moving averages* (SMAs). The idea is that the trader should go long on a stock (or financial instrument in general) when the shorter-term SMA is above the longer-term SMA and should go short when the opposite holds true. The concepts can be made precise with `pandas` and the capabilities of the `DataFrame` object.

Rolling statistics are generally only calculated when there is enough data given the `window` parameter specification. As Figure 8-6 shows, the SMA time series only start at the day for which there is enough data given the specific parameterization:

```
In [37]: data['SMA1'] = data[sym].rolling(window=42).mean()  ❶
```

```
In [38]: data['SMA2'] = data[sym].rolling(window=252).mean()  ❷

In [39]: data[[sym, 'SMA1', 'SMA2']].tail()
Out[39]:             AAPL.O        SMA1        SMA2
         Date
         2018-06-25  182.17  185.606190  168.265556
         2018-06-26  184.43  186.087381  168.418770
         2018-06-27  184.16  186.607381  168.579206
         2018-06-28  185.50  187.089286  168.736627
         2018-06-29  185.11  187.470476  168.901032

In [40]: data[[sym, 'SMA1', 'SMA2']].plot(figsize=(10, 6));  ❸
```

❶ Calculates the values for the shorter-term SMA.

❷ Calculates the values for the longer-term SMA.

❸ Visualizes the stock price data plus the two SMA time series.

Figure 8-6. Apple stock price and two simple moving averages

In this context, the SMAs are only a means to an end. They are used to derive positions to implement a trading strategy. Figure 8-7 visualizes a long position by a value of `1` and a short position by a value of `-1`. The change in the position is triggered (visually) by a crossover of the two lines representing the SMA time series:

```
In [41]: data.dropna(inplace=True)  ❶
```

```
In [42]: data['positions'] = np.where(data['SMA1'] > data['SMA2'],  ❷
                                      1,  ❸
                                      -1)  ❹

In [43]: ax = data[[sym, 'SMA1', 'SMA2', 'positions']].plot(figsize=(10, 6),
                                                  secondary_y='positions')
         ax.get_legend().set_bbox_to_anchor((0.25, 0.85));
```

❶ Only complete data rows are kept.

❷ If the shorter-term SMA value is greater than the longer-term one …

❸ … go long on the stock (put a `1`).

❹ Otherwise, go short on the stock (put a `-1`).

Figure 8-7. Apple stock price, two simple moving averages and positions

The trading strategy implicitly derived here only leads to a few trades per se: only when the position value changes (i.e., a crossover happens) does a trade take place. Including opening and closing trades, this would add up to just six trades in total.

Correlation Analysis

As a further illustration of how to work with `pandas` and financial time series data, consider the case of the S&P 500 stock index and the VIX volatility index. It is a stylized fact that when the S&P 500 rises, the VIX falls in general, and vice versa. This is about *correlation* and not *causation*. This section shows how to come up with some supporting statistical evidence for the stylized fact that the S&P 500 and the VIX are (highly) negatively correlated.[4]

The Data

The data set now consists of two financial times series, both visualized in Figure 8-8:

```
In [44]: raw = pd.read_csv('../../source/tr_eikon_eod_data.csv',
                           index_col=0, parse_dates=True)  ❶

In [45]: data = raw[['.SPX', '.VIX']].dropna()

In [46]: data.tail()
Out[46]:                .SPX   .VIX
         Date
         2018-06-25  2717.07  17.33
         2018-06-26  2723.06  15.92
         2018-06-27  2699.63  17.91
         2018-06-28  2716.31  16.85
         2018-06-29  2718.37  16.09

In [47]: data.plot(subplots=True, figsize=(10, 6));
```

❶ Reads the EOD data (originally from the Thomson Reuters Eikon Data API) from a CSV file.

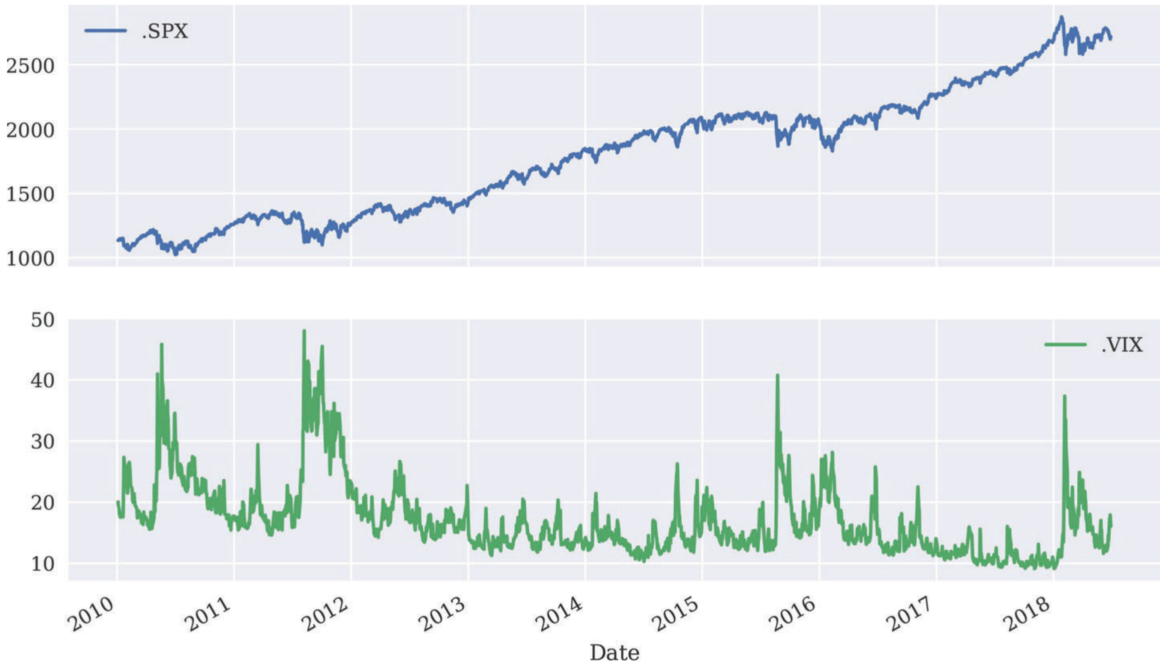

Figure 8-8. S&P 500 and VIX time series data (different subplots)

When plotting (parts of) the two time series in a single plot and with adjusted scalings, the stylized fact of negative correlation between the two indices becomes evident through simple visual inspection (Figure 8-9):

```
In [48]: data.loc[:'2012-12-31'].plot(secondary_y='.VIX', figsize=(10, 6));  ❶
```

❶ `.loc[:DATE]` selects the data until the given value `DATE`.

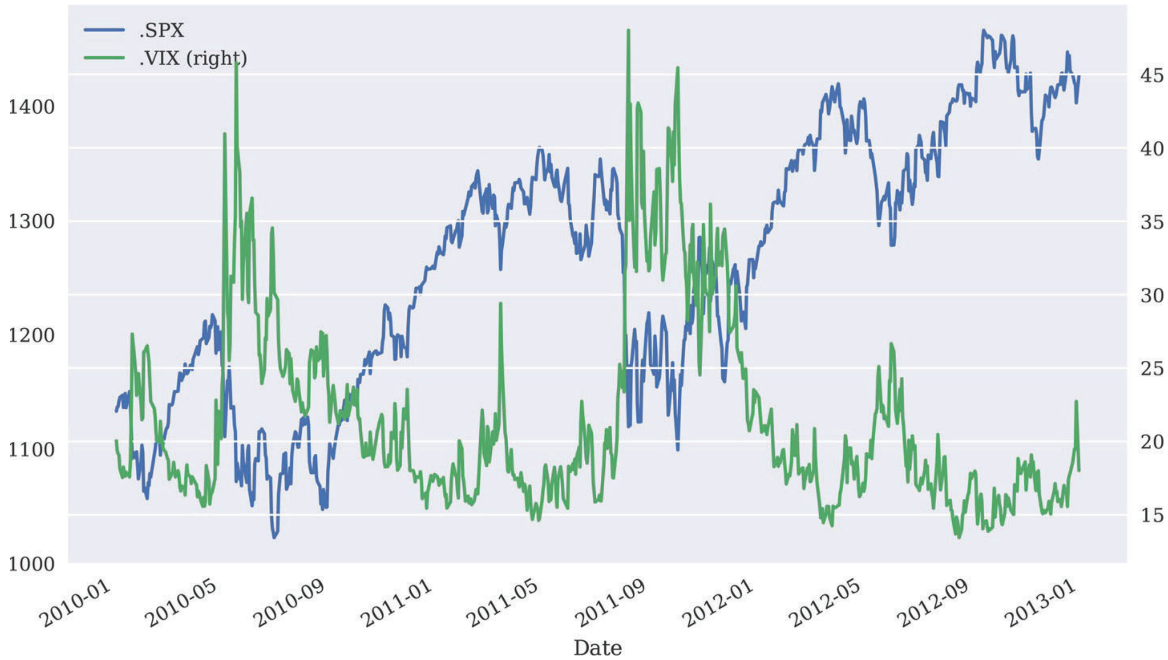

Figure 8-9. S&P 500 and VIX time series data (same plot)

Logarithmic Returns

As pointed out earlier, statistical analysis in general relies on returns instead of absolute changes or even absolute values. Therefore, we'll calculate log returns first before any further analysis takes place. Figure 8-10 shows the high variability of the log returns over time. For both indices so-called "volatility clusters" can be spotted. In general, periods of high volatility in the stock index are accompanied by the same phenomena in the volatility index:

```
In [49]: rets = np.log(data / data.shift(1))

In [50]: rets.head()
Out[50]:                  .SPX      .VIX
         Date
         2010-01-04        NaN       NaN
         2010-01-05   0.003111 -0.035038
         2010-01-06   0.000545 -0.009868
         2010-01-07   0.003993 -0.005233
         2010-01-08   0.002878 -0.050024

In [51]: rets.dropna(inplace=True)

In [52]: rets.plot(subplots=True, figsize=(10, 6));
```

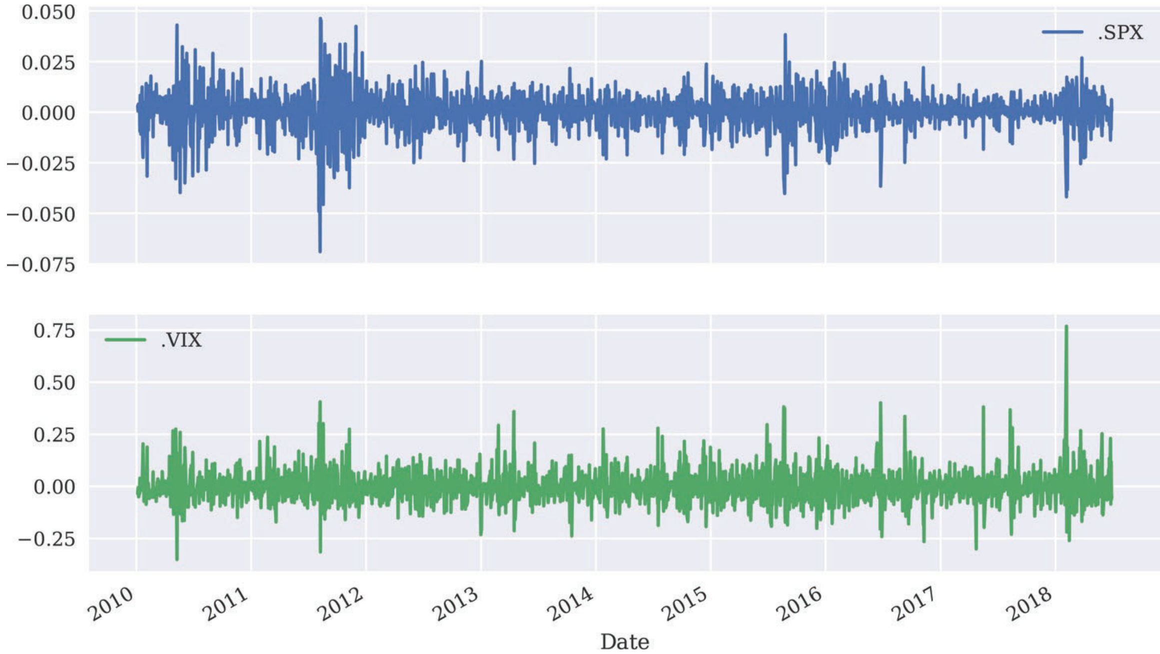

Figure 8-10. Log returns of the S&P 500 and VIX over time

In such a context, the pandas `scatter_matrix()` plotting function comes in handy for visualizations. It plots the log returns of the two series against each other, and one can add either a histogram or a kernel density estimator (KDE) on the diagonal (see Figure 8-11):

```
In [53]: pd.plotting.scatter_matrix(rets,  ❶
                                    alpha=0.2,  ❷
                                    diagonal='hist',  ❸
                                    hist_kwds={'bins': 35},  ❹
                                    figsize=(10, 6));
```

❶ The data set to be plotted.

❷ The `alpha` parameter for the opacity of the dots.

❸ What to place on the diagonal; here: a histogram of the column data.

❹ Keywords to be passed to the histogram plotting function.

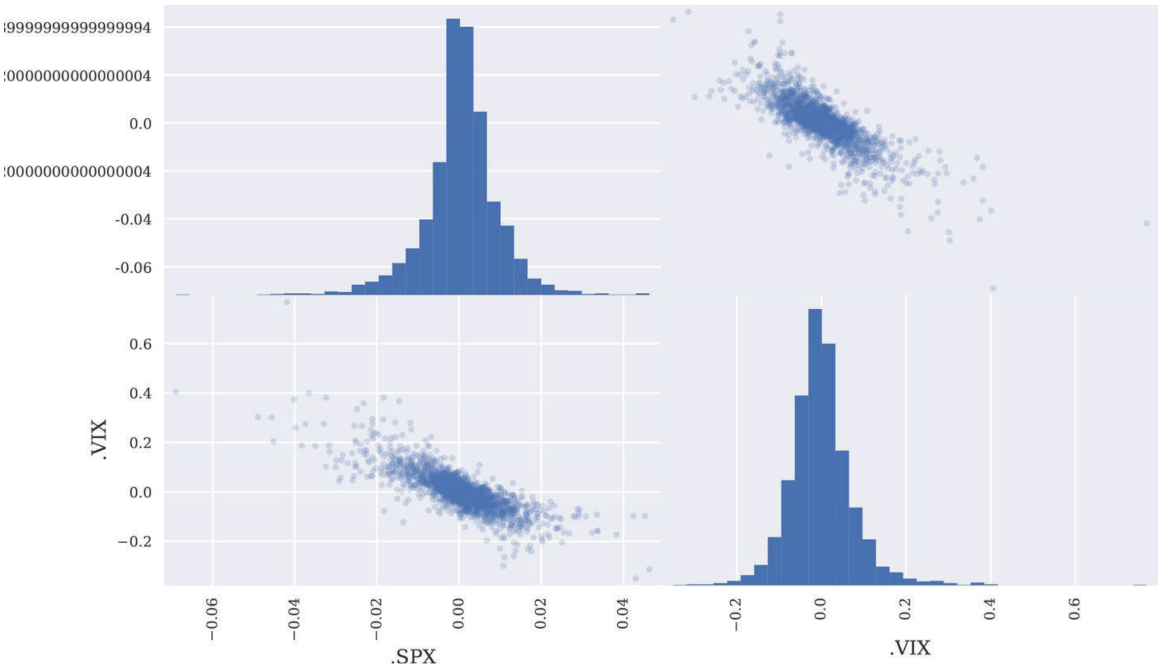

Figure 8-11. Log returns of the S&P 500 and VIX as a scatter matrix

OLS Regression

With all these preparations, an ordinary least-squares (OLS) regression analysis is convenient to implement. Figure 8-12 shows a scatter plot of the log returns and the linear regression line through the cloud of dots. The slope is obviously negative, providing support for the stylized fact about the negative correlation between the two indices:

```
In [54]: reg = np.polyfit(rets['.SPX'], rets['.VIX'], deg=1)  ❶

In [55]: ax = rets.plot(kind='scatter', x='.SPX', y='.VIX', figsize=(10, 6))  ❷
         ax.plot(rets['.SPX'], np.polyval(reg, rets['.SPX']), 'r', lw=2);  ❸
```

❶ This implements a linear OLS regression.

❷ This plots the log returns as a scatter plot …

❸ … to which the linear regression line is added.

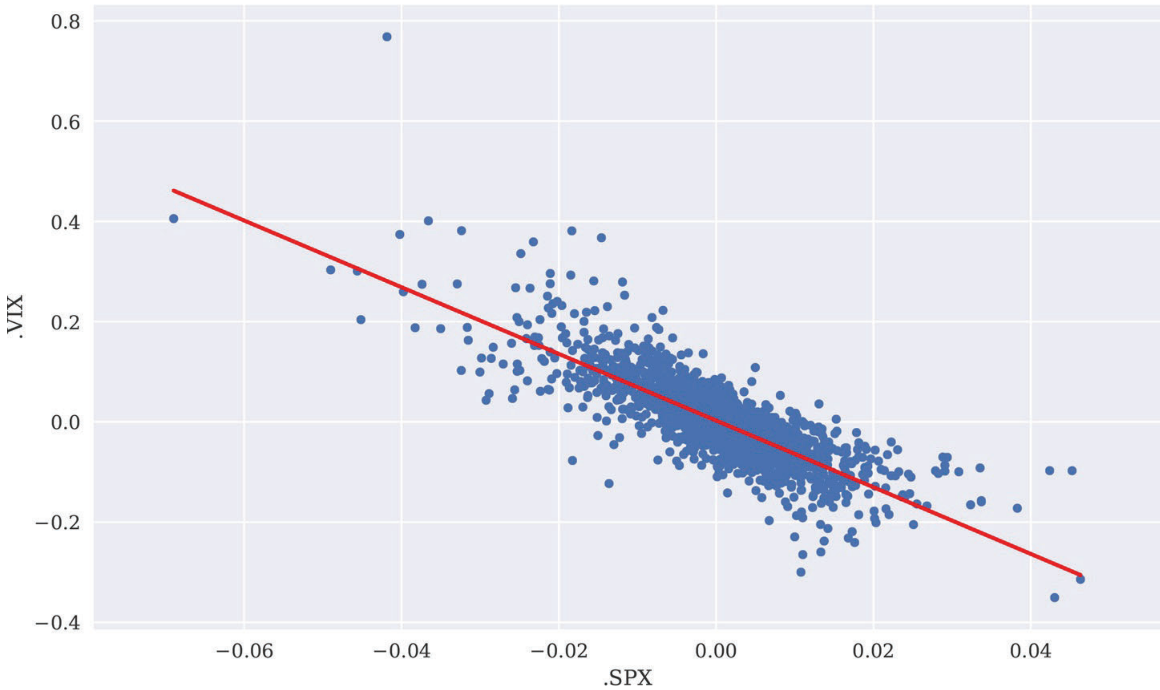

Figure 8-12. Log returns of the S&P 500 and VIX as a scatter matrix

Correlation

Finally, we consider correlation measures directly. Two such measures are considered: a static one taking into account the complete data set and a rolling one showing the correlation for a fixed window over time. Figure 8-13 illustrates that the correlation indeed varies over time but that it is always, given the parameterization, negative. This provides strong support for the stylized fact that the S&P 500 and the VIX indices are (strongly) negatively correlated:

```
In [56]: rets.corr()  ❶
Out[56]:           .SPX      .VIX
         .SPX  1.000000 -0.804382
         .VIX -0.804382  1.000000

In [57]: ax = rets['.SPX'].rolling(window=252).corr(
                           rets['.VIX']).plot(figsize=(10, 6))  ❷
         ax.axhline(rets.corr().iloc[0, 1], c='r');  ❸
```

❶ The correlation matrix for the whole `DataFrame`.

❷ This plots the rolling correlation over time …

❸ … and adds the static value to the plot as horizontal line.

Figure 8-13. Correlation between S&P 500 and VIX (static and rolling)

High-Frequency Data

This chapter is about financial time series analysis with `pandas`. Tick data sets are a special case of financial time series. Frankly, they can be handled more or less in the same ways as, for instance, the EOD data set used throughout this chapter so far. Importing such data sets also is quite fast in general with `pandas`. The data set used comprises 17,352 data rows (see also Figure 8-14):

```
In [59]: %%time
         # data from FXCM Forex Capital Markets Ltd.
         tick = pd.read_csv('../../source/fxcm_eur_usd_tick_data.csv',
                                      index_col=0, parse_dates=True)
         CPU times: user 1.07 s, sys: 149 ms, total: 1.22 s
         Wall time: 1.16 s

In [60]: tick.info()
         <class 'pandas.core.frame.DataFrame'>
         DatetimeIndex: 461357 entries, 2018-06-29 00:00:00.082000 to 2018-06-29
          20:59:00.607000
         Data columns (total 2 columns):
         Bid     461357 non-null float64
         Ask     461357 non-null float64
         dtypes: float64(2)
         memory usage: 10.6 MB
```

```
In [61]: tick['Mid'] = tick.mean(axis=1)  ❶

In [62]: tick['Mid'].plot(figsize=(10, 6));
```

❶ Calculates the `Mid` price for every data row.

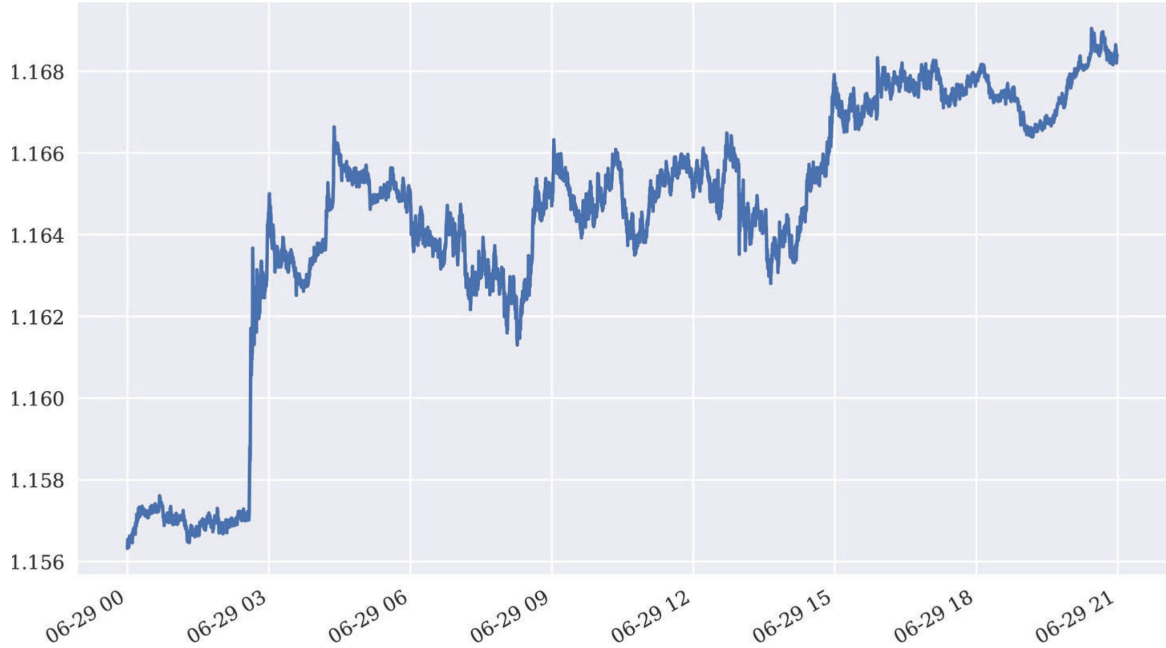

Figure 8-14. Tick data for EUR/USD exchange rate

Working with tick data is generally a scenario where resampling of financial time series data is needed. The code that follows resamples the tick data to five-minute bar data (see Figure 8-15), which can then be used, for example, to backtest algorithmic trading strategies or to implement a technical analysis:

```
In [63]: tick_resam = tick.resample(rule='5min', label='right').last()

In [64]: tick_resam.head()
Out[64]:                          Bid      Ask       Mid
         2018-06-29 00:05:00  1.15649  1.15651  1.156500
         2018-06-29 00:10:00  1.15671  1.15672  1.156715
         2018-06-29 00:15:00  1.15725  1.15727  1.157260
         2018-06-29 00:20:00  1.15720  1.15722  1.157210
         2018-06-29 00:25:00  1.15711  1.15712  1.157115

In [65]: tick_resam['Mid'].plot(figsize=(10, 6));
```

Figure 8-15. Five-minute bar data for EUR/USD exchange rate

Conclusion

This chapter deals with financial time series, probably the most important data type in the financial field. `pandas` is a powerful package to deal with such data sets, allowing not only for efficient data analyses but also easy visualizations, for instance. `pandas` is also helpful in reading such data sets from different sources as well as in exporting the data sets to different technical file formats. This is illustrated in the subsequent chapter.

Further Resources

Good references in book form for the topics covered in this chapter are:

- McKinney, Wes (2017). *Python for Data Analysis*. Sebastopol, CA: O'Reilly.
- VanderPlas, Jake (2016). *Python Data Science Handbook*. Sebastopol, CA: O'Reilly.