O'REILLY®

2nd Edition

Python for Finance

MASTERING DATA-DRIVEN FINANCE

Yves Hilpisch

Python for Finance

Mastering Data-Driven Finance

SECOND EDITION

Yves Hilpisch

Beijing • Boston • Farnham • Sebastopol • Tokyo

Python for Finance

by Yves Hilpisch

Printed in Canada.

Published by O’Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

Editors: Susan Conant and Jeff Bleiel	Indexer: Judith McConville
Production Editor: Kristen Brown	Interior Designer: David Futato
Copyeditor: Rachel Head	Cover Designer: Karen Montgomery
Proofreader: Kim Cofer	Illustrator: Rebecca Demarest

December 2014: First Edition

December 2018: Second Edition

Revision History for the Second Edition

- 2018-11-29: First Release
- 2019-01-18: Second Release

See http://oreilly.com/catalog/errata.csp?isbn=9781492024330 for release details.

978-1-492-02433-0

[MBP]

Part II. Mastering the Basics

This part of the book is concerned with the basics of Python programming. The topics covered in this part are fundamental for all other chapters to follow in subsequent parts and for Python usage in general.

The chapters are organized according to certain topics such that they can be used as a reference to which the reader can come to look up examples and details related to the topic of interest:

- Chapter 3 focuses on Python data types and structures.
- Chapter 4 is about `NumPy` and its `ndarray` class.
- Chapter 5 is about `pandas` and its `DataFrame` class.
- Chapter 6 discusses object-oriented programming (OOP) with Python.

Chapter 3. Data Types and Structures

Bad programmers worry about the code. Good programmers worry about data structures and their relationships.

—Linus Torvalds

This chapter introduces the basic data types and data structures of Python, and is organized as follows:

"Basic Data Types"

The first section introduces basic data types such as `int`, `float`, `bool`, and `str`.

"Basic Data Structures"

The second section introduces the fundamental data structures of Python (e.g., `list` objects) and illustrates, among other things, control structures, functional programming approaches, and anonymous functions.

The aim of this chapter is to provide a general introduction to Python specifics when it comes to data types and structures. The reader equipped with a background from another programing language, say C or Matlab, should be able to easily grasp the differences that Python usage might bring along. The topics and idioms introduced here are important and fundamental for the chapters to come.

The chapter covers the following data types and structures:

Object type	Meaning	Used for
`int`	Integer value	Natural numbers
`float`	Floating-point number	Real numbers
`bool`	Boolean value	Something true or false
`str`	String object	Character, word, text

`tuple`	Immutable container	Fixed set of objects, record
`list`	Mutable container	Changing set of objects
`dict`	Mutable container	Key-value store
`set`	Mutable container	Collection of unique objects

Basic Data Types

Python is a *dynamically typed* language, which means that the Python interpreter infers the type of an object at runtime. In comparison, compiled languages like C are generally *statically typed*. In these cases, the type of an object has to be specified for the object before compile time.[1]

Integers

One of the most fundamental data types is the integer, or `int`:

```
In [1]: a = 10
        type(a)
Out[1]: int
```

The built-in function `type` provides type information for all objects with standard and built-in types as well as for newly created classes and objects. In the latter case, the information provided depends on the description the programmer has stored with the class. There is a saying that “everything in Python is an object.” This means, for example, that even simple objects like the `int` object just defined have built-in methods. For example, one can get the number of bits needed to represent the `int` object in memory by calling the method `bit_length()`:

```
In [2]: a.bit_length()
Out[2]: 4
```

The number of bits needed increases the higher the integer value is that one assigns to the object:

```
In [3]: a = 100000
        a.bit_length()
Out[3]: 17
```

In general, there are so many different methods that it is hard to memorize all methods of all classes and objects. Advanced Python environments like IPython provide tab completion capabilities that show all the methods attached to an object. You simply type the object name followed by a dot (e.g., `a.`) and then press the Tab key. This then provides a collection of methods you can call on the object. Alternatively, the Python built-in function `dir` gives a complete list of the attributes and methods of any object.

A specialty of Python is that integers can be arbitrarily large. Consider, for example, the googol number 10^{100}. Python has no problem with such large numbers:

```
In [4]: googol = 10 ** 100
        googol
Out[4]: 10000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
        00000000000000000000000000

In [5]: googol.bit_length()
Out[5]: 333
```

> **LARGE INTEGERS**
>
> Python integers can be arbitrarily large. The interpreter simply uses as many bits/bytes as needed to represent the numbers.

Arithmetic operations on integers are also easy to implement:

```
In [6]: 1 + 4
Out[6]: 5

In [7]: 1 / 4
Out[7]: 0.25

In [8]: type(1 / 4)
Out[8]: float
```

Floats

The last expression returns the mathematically correct result of 0.25,[2] which gives rise to the next basic data type, the `float`. Adding a dot to an integer value, like in `1.` or `1.0`, causes Python to interpret the object as a `float`. Expressions involving a `float` also return a `float` object in general:[3]

```
In [9]: 1.6 / 4
Out[9]: 0.4

In [10]: type (1.6 / 4)
Out[10]: float
```

A `float` is a bit more involved in that the computerized representation of rational or real numbers is in general not exact and depends on the specific technical approach taken. To illustrate what this implies, let us define another `float` object, `b`. `float` objects like this one are always represented internally up to a certain degree of accuracy only. This becomes evident when adding `0.1` to `b`:

```
In [11]: b = 0.35
         type(b)
Out[11]: float

In [12]: b + 0.1
Out[12]: 0.44999999999999996
```

The reason for this is that `float` objects are internally represented in binary format; that is, a decimal number $0 < n < 1$ is represented by a series of the form $n = \frac{x}{2} + \frac{y}{4} + \frac{z}{8} + \ldots$. For certain floating-point numbers the binary representation might involve a large number of elements or might even be an infinite series. However, given a fixed number of bits used to represent such a number—i.e., a fixed number of terms in the representation series—inaccuracies are the consequence. Other numbers can be represented *perfectly* and are therefore stored exactly even with a finite number of bits available. Consider the following example:

```
In [13]: c = 0.5
         c.as_integer_ratio()
```

```
Out[13]: (1, 2)
```

One-half, i.e., 0.5, is stored exactly because it has an exact (finite) binary representation as $0.5 = \frac{1}{2}$. However, for `b = 0.35` one gets something different than the expected rational number $0.35 = \frac{7}{20}$:

```
In [14]: b.as_integer_ratio()
Out[14]: (3152519739159347, 9007199254740992)
```

The precision is dependent on the number of bits used to represent the number. In general, all platforms that Python runs on use the IEEE 754 double-precision standard—i.e., 64 bits—for internal representation. This translates into a 15-digit relative accuracy.

Since this topic is of high importance for several application areas in finance, it is sometimes necessary to ensure the exact, or at least best possible, representation of numbers. For example, the issue can be of importance when summing over a large set of numbers. In such a situation, a certain kind and/or magnitude of representation error might, in aggregate, lead to significant deviations from a benchmark value.

The module `decimal` provides an arbitrary-precision object for floating-point numbers and several options to address precision issues when working with such numbers:

```
In [15]: import decimal
         from decimal import Decimal

In [16]: decimal.getcontext()
Out[16]: Context(prec=28, rounding=ROUND_HALF_EVEN, Emin=-999999, Emax=999999,
          capitals=1, clamp=0, flags=[], traps=[InvalidOperation, DivisionByZero,
          Overflow])

In [17]: d = Decimal(1) / Decimal (11)
         d
Out[17]: Decimal('0.090909090909090909090909090909091')
```

One can change the precision of the representation by changing the respective attribute value of the `Context` object:

```
In [18]: decimal.getcontext().prec = 4  ❶

In [19]: e = Decimal(1) / Decimal (11)
         e
Out[19]: Decimal('0.09091')

In [20]: decimal.getcontext().prec = 50  ❷

In [21]: f = Decimal(1) / Decimal (11)
         f
Out[21]: Decimal('0.090909090909090909090909090909090909090909090909091')
```

❶ Lower precision than default.

❷ Higher precision than default.

If needed, the precision can in this way be adjusted to the exact problem at hand and one can operate with floating-point objects that exhibit different degrees of accuracy:

```
In [22]: g = d + e + f
         g
Out[22]: Decimal('0.272728181818181818181818181819090909090909090909091')
```

> **ARBITRARY-PRECISION FLOATS**
>
> The module `decimal` provides an arbitrary-precision floating-point number object. In finance, it might sometimes be necessary to ensure high precision and to go beyond the 64-bit double-precision standard.

Booleans

In programming, evaluating a comparison or logical expression (such as `4 > 3`, `4.5 <= 3.25` or `(4 > 3) and (3 > 2)`) yields one of `True` or `False` as output, two important Python keywords. Others are, for example, `def`, `for`, and `if`. A complete list of Python keywords is available in the `keyword` module:

```
In [23]: import keyword

In [24]: keyword.kwlist
Out[24]: ['False',
          'None',
          'True',
```

```
 'and',
 'as',
 'assert',
 'async',
 'await',
 'break',
 'class',
 'continue',
 'def',
 'del',
 'elif',
 'else',
 'except',
 'finally',
 'for',
 'from',
 'global',
 'if',
 'import',
 'in',
 'is',
 'lambda',
 'nonlocal',
 'not',
 'or',
 'pass',
 'raise',
 'return',
 'try',
 'while',
 'with',
 'yield']
```

`True` and `False` are of data type `bool`, standing for *Boolean value*. The following code shows Python's *comparison* operators applied to the same operands with the resulting `bool` objects:

```
In [25]: 4 > 3  ❶
Out[25]: True

In [26]: type(4 > 3)
Out[26]: bool

In [27]: type(False)
Out[27]: bool

In [28]: 4 >= 3  ❷
```

```
Out[28]: True

In [29]: 4 < 3  ❸
Out[29]: False

In [30]: 4 <= 3  ❹
Out[30]: False

In [31]: 4 == 3  ❺
Out[31]: False

In [32]: 4 != 3  ❻
Out[32]: True
```

❶ Is greater.
❷ Is greater or equal.
❸ Is smaller.
❹ Is smaller or equal.
❺ Is equal.
❻ Is not equal.

Often, *logical* operators are applied on `bool` objects, which in turn yields another `bool` object:

```
In [33]: True and True
Out[33]: True

In [34]: True and False
Out[34]: False

In [35]: False and False
Out[35]: False

In [36]: True or True
Out[36]: True

In [37]: True or False
Out[37]: True

In [38]: False or False
Out[38]: False

In [39]: not True
Out[39]: False
```

```
In [40]: not False
Out[40]: True
```

Of course, both types of operators are often combined:

```
In [41]: (4 > 3) and (2 > 3)
Out[41]: False

In [42]: (4 == 3) or (2 != 3)
Out[42]: True

In [43]: not (4 != 4)
Out[43]: True

In [44]: (not (4 != 4)) and (2 == 3)
Out[44]: False
```

One major application area is to control the code flow via other Python keywords, such as `if` or `while` (more examples later in the chapter):

```
In [45]: if 4 > 3:  ❶
             print('condition true')  ❷
         condition true

In [46]: i = 0  ❸
         while i < 4:  ❹
             print('condition true, i = ', i)  ❺
             i += 1  ❻
         condition true, i =  0
         condition true, i =  1
         condition true, i =  2
         condition true, i =  3
```

❶ If condition holds true, execute code to follow.
❷ The code to be executed if condition holds true.
❸ Initializes the parameter `i` with `0`.
❹ As long as the condition holds true, execute and repeat the code to follow.
❺ Prints a text and the value of parameter `i`.
❻ Increases the parameter value by 1; `i += 1` is the same as `i = i + 1`.

Numerically, Python attaches a value of 0 to `False` and a value of 1 to `True`. When transforming numbers to `bool` objects via the `bool()` function, a 0 gives

`False` while all other numbers give `True`:

```
In [47]: int(True)
Out[47]: 1

In [48]: int(False)
Out[48]: 0

In [49]: float(True)
Out[49]: 1.0

In [50]: float(False)
Out[50]: 0.0

In [51]: bool(0)
Out[51]: False

In [52]: bool(0.0)
Out[52]: False

In [53]: bool(1)
Out[53]: True

In [54]: bool(10.5)
Out[54]: True

In [55]: bool(-2)
Out[55]: True
```

Strings

Now that natural and floating-point numbers can be represented, this subsection turns to text. The basic data type to represent text in Python is `str`. The `str` object has a number of helpful built-in methods. In fact, Python is generally considered to be a good choice when it comes to working with texts and text files of any kind and any size. A `str` object is generally defined by single or double quotation marks or by converting another object using the `str()` function (i.e., using the object's standard or user-defined `str` representation):

```
In [56]: t = 'this is a string object'
```

With regard to the built-in methods, you can, for example, capitalize the first word in this object:

```
In [57]: t.capitalize()
Out[57]: 'This is a string object'
```

Or you can split it into its single-word components to get a `list` object of all the words (more on `list` objects later):

```
In [58]: t.split()
Out[58]: ['this', 'is', 'a', 'string', 'object']
```

You can also search for a word and get the position (i.e., index value) of the first letter of the word back in a successful case:

```
In [59]: t.find('string')
Out[59]: 10
```

If the word is not in the `str` object, the method returns `-1`:

```
In [60]: t.find('Python')
Out[60]: -1
```

Replacing characters in a string is a typical task that is easily accomplished with the `replace()` method:

```
In [61]: t.replace(' ', '|')
Out[61]: 'this|is|a|string|object'
```

The stripping of strings—i.e., deletion of certain leading/lagging characters—is also often necessary:

```
In [62]: 'http://www.python.org'.strip('htp:/')
Out[62]: 'www.python.org'
```

Table 3-1 lists a number of helpful methods of the `str` object.

Table 3-1. Selected string methods

Method	Arguments	Returns/result
`capitalize`	`()`	Copy of the string with first letter capitalized

`count`	(*sub*[, *start*[, *end*]])	Count of the number of occurrences of substring
`decode`	([*encoding*[, *errors*]])	Decoded version of the string, using *encoding* (e.g., UTF-8)
`encode`	([*encoding*+[, *errors*]])	Encoded version of the string
`find`	(*sub*[, *start*[, *end*]])	(Lowest) index where substring is found
`join`	(*seq*)	Concatenation of strings in sequence *seq*
`replace`	(*old*, *new*[, *count*])	Replaces *old* by *new* the first *count* times
`split`	([*sep*[, *maxsplit*]])	List of words in string with *sep* as separator
`splitlines`	([*keepends*])	Separated lines with line ends/breaks if *keepends* is `True`
`strip`	(*chars*)	Copy of string with leading/lagging characters in *chars* removed
`upper`	()	Copy with all letters capitalized

UNICODE STRINGS

A fundamental change from Python 2.7 (used for the first edition of the book) to Python 3.7 (used for this second edition) is the encoding and decoding of string objects and the introduction of Unicode. This chapter does not go into the many details important in this context; for the purposes of this book, which mainly deals with numerical data and standard strings containing English words, this omission seems justified.

Excursion: Printing and String Replacements

Printing `str` objects or string representations of other Python objects is usually accomplished by the `print()` function:

```
In [63]: print('Python for Finance')  ❶
         Python for Finance

In [64]: print(t)  ❷
         this is a string object

In [65]: i = 0
```

```
            while i < 4:
                print(i)  ❸
                i += 1
            0
            1
            2
            3

In [66]: i = 0
            while i < 4:
                print(i, end='|')  ❹
                i += 1
            0|1|2|3|
```

❶ Prints a `str` object.

❷ Prints a `str` object referenced by a variable name.

❸ Prints the string representation of an `int` object.

❹ Specifies the final character(s) when printing; default is a line break (`\n`) as seen before.

Python offers powerful string replacement operations. There is the old way, via the `%` character, and the new way, via curly braces (`{}`) and `format()`. Both are still applied in practice. This section cannot provide an exhaustive illustration of all options, but the following code snippets show some important ones. First, the *old* way of doing it:

```
In [67]: 'this is an integer %d' % 15  ❶
Out[67]: 'this is an integer 15'

In [68]: 'this is an integer %4d' % 15  ❷
Out[68]: 'this is an integer   15'

In [69]: 'this is an integer %04d' % 15  ❸
Out[69]: 'this is an integer 0015'

In [70]: 'this is a float %f' % 15.3456  ❹
Out[70]: 'this is a float 15.345600'

In [71]: 'this is a float %.2f' % 15.3456  ❺
Out[71]: 'this is a float 15.35'

In [72]: 'this is a float %8f' % 15.3456  ❻
Out[72]: 'this is a float 15.345600'

In [73]: 'this is a float %8.2f' % 15.3456  ❼
```

```
Out[73]: 'this is a float    15.35'

In [74]: 'this is a float %08.2f' % 15.3456  ❽
Out[74]: 'this is a float 00015.35'

In [75]: 'this is a string %s' % 'Python'  ❾
Out[75]: 'this is a string Python'

In [76]: 'this is a string %10s' % 'Python'  ❿
Out[76]: 'this is a string     Python'
```

❶ `int` object replacement.
❷ With fixed number of characters.
❸ With leading zeros if necessary.
❹ `float` object replacement.
❺ With fixed number of decimals.
❻ With fixed number of characters (and filled-up decimals).
❼ With fixed number of characters and decimals …
❽ … and leading zeros if necessary.
❾ `str` object replacement.
❿ With fixed number of characters.

Now, here are the same examples implemented in the *new* way. Notice the slight differences in the output in some places:

```
In [77]: 'this is an integer {:d}'.format(15)
Out[77]: 'this is an integer 15'

In [78]: 'this is an integer {:4d}'.format(15)
Out[78]: 'this is an integer   15'

In [79]: 'this is an integer {:04d}'.format(15)
Out[79]: 'this is an integer 0015'

In [80]: 'this is a float {:f}'.format(15.3456)
Out[80]: 'this is a float 15.345600'

In [81]: 'this is a float {:.2f}'.format(15.3456)
Out[81]: 'this is a float 15.35'

In [82]: 'this is a float {:8f}'.format(15.3456)
Out[82]: 'this is a float 15.345600'
```

```
In [83]: 'this is a float {:8.2f}'.format(15.3456)
Out[83]: 'this is a float    15.35'

In [84]: 'this is a float {:08.2f}'.format(15.3456)
Out[84]: 'this is a float 00015.35'

In [85]: 'this is a string {:s}'.format('Python')
Out[85]: 'this is a string Python'

In [86]: 'this is a string {:10s}'.format('Python')
Out[86]: 'this is a string Python    '
```

String replacements are particularly useful in the context of multiple printing operations where the printed data is updated, for instance, during a `while` loop:

```
In [87]: i = 0
         while i < 4:
             print('the number is %d' % i)
             i += 1
         the number is 0
         the number is 1
         the number is 2
         the number is 3

In [88]: i = 0
         while i < 4:
             print('the number is {:d}'.format(i))
             i += 1
         the number is 0
         the number is 1
         the number is 2
         the number is 3
```

Excursion: Regular Expressions

A powerful tool when working with `str` objects is *regular expressions*. Python provides such functionality in the module `re`:

```
In [89]: import re
```

Suppose a financial analyst is faced with a large text file, such as a CSV file, which contains certain time series and respective date-time information. More often than not, this information is delivered in a format that Python cannot interpret directly. However, the date-time information can generally be described

by a regular expression. Consider the following `str` object, containing three date-time elements, three integers, and three strings. Note that triple quotation marks allow the definition of `str` objects over multiple rows:

```
In [90]: series = """
         '01/18/2014 13:00:00', 100, '1st';
         '01/18/2014 13:30:00', 110, '2nd';
         '01/18/2014 14:00:00', 120, '3rd'
         """
```

The following regular expression describes the format of the date-time information provided in the `str` object:[4]

```
In [91]: dt = re.compile("'[0-9/:\s]+'")  # datetime
```

Equipped with this regular expression, one can go on and find all the date-time elements. In general, applying regular expressions to `str` objects also leads to performance improvements for typical parsing tasks:

```
In [92]: result = dt.findall(series)
         result
Out[92]: ["'01/18/2014 13:00:00'", "'01/18/2014 13:30:00'", "'01/18/2014
          14:00:00'"]
```

> **REGULAR EXPRESSIONS**
>
> When parsing `str` objects, consider using regular expressions, which can bring both convenience and performance to such operations.

The resulting `str` objects can then be parsed to generate Python `datetime` objects (see Appendix A for an overview of handling date and time data with Python). To parse the `str` objects containing the date-time information, one needs to provide information of how to parse them—again as a `str` object:

```
In [93]: from datetime import datetime
         pydt = datetime.strptime(result[0].replace("'", ""),
                                  '%m/%d/%Y %H:%M:%S')
         pydt
```

```
Out[93]: datetime.datetime(2014, 1, 18, 13, 0)

In [94]: print(pydt)
         2014-01-18 13:00:00

In [95]: print(type(pydt))
         <class 'datetime.datetime'>
```

Later chapters provide more information on date-time data, the handling of such data, and `datetime` objects and their methods. This is just meant to be a teaser for this important topic in finance.

Basic Data Structures

As a general rule, data structures are objects that contain a possibly large number of other objects. Among those that Python provides as built-in structures are:

`tuple`

: An immutable collection of arbitrary objects; only a few methods available

`list`

: A mutable collection of arbitrary objects; many methods available

`dict`

: A key-value store object

`set`

: An unordered collection object for other *unique* objects

Tuples

A `tuple` is an advanced data structure, yet it's still quite simple and limited in its applications. It is defined by providing objects in parentheses:

```
In [96]: t = (1, 2.5, 'data')
         type(t)
Out[96]: tuple
```

You can even drop the parentheses and provide multiple objects, just separated by commas:

```
In [97]: t = 1, 2.5, 'data'
         type(t)
Out[97]: tuple
```

Like almost all data structures in Python the `tuple` has a built-in index, with the help of which you can retrieve single or multiple elements of the `tuple`. It is important to remember that Python uses *zero-based numbering*, such that the third element of a `tuple` is at index position 2:

```
In [98]: t[2]
Out[98]: 'data'

In [99]: type(t[2])
Out[99]: str
```

> **ZERO-BASED NUMBERING**
>
> In contrast to some other programming languages like Matlab, Python uses zero-based numbering schemes. For example, the first element of a `tuple` object has index value 0.

There are only two special methods that this object type provides: `count()` and `index()`. The first counts the number of occurrences of a certain object and the second gives the index value of the first appearance of it:

```
In [100]: t.count('data')
Out[100]: 1

In [101]: t.index(1)
Out[101]: 0
```

`tuple` objects are *immutable* objects. This means that they, once defined, cannot be changed easily.

Lists

Objects of type `list` are much more flexible and powerful in comparison to `tuple` objects. From a finance point of view, you can achieve a lot working only with `list` objects, such as storing stock price quotes and appending new data. A `list` object is defined through brackets and the basic capabilities and behaviors are similar to those of `tuple` objects:

```
In [102]: l = [1, 2.5, 'data']
          l[2]
Out[102]: 'data'
```

`list` objects can also be defined or converted by using the function `list()`. The following code generates a new `list` object by converting the `tuple` object from the previous example:

```
In [103]: l = list(t)
          l
Out[103]: [1, 2.5, 'data']

In [104]: type(l)
Out[104]: list
```

In addition to the characteristics of `tuple` objects, `list` objects are also expandable and reducible via different methods. In other words, whereas `str` and `tuple` objects are *immutable* sequence objects (with indexes) that cannot be changed once created, `list` objects are *mutable* and can be changed via different operations. You can append `list` objects to an existing `list` object, and more:

```
In [105]: l.append([4, 3])  ❶
          l
Out[105]: [1, 2.5, 'data', [4, 3]]

In [106]: l.extend([1.0, 1.5, 2.0])  ❷
          l
Out[106]: [1, 2.5, 'data', [4, 3], 1.0, 1.5, 2.0]

In [107]: l.insert(1, 'insert')  ❸
          l
Out[107]: [1, 'insert', 2.5, 'data', [4, 3], 1.0, 1.5, 2.0]

In [108]: l.remove('data')  ❹
          l
```

```
Out[108]: [1, 'insert', 2.5, [4, 3], 1.0, 1.5, 2.0]

In [109]: p = l.pop(3)  ❺
          print(l, p)
          [1, 'insert', 2.5, 1.0, 1.5, 2.0] [4, 3]
```

❶ Append `list` object at the end.
❷ Append elements of the `list` object.
❸ Insert object before index position.
❹ Remove first occurrence of object.
❺ Remove and return object at index position.

Slicing is also easily accomplished. Here, *slicing* refers to an operation that breaks down a data set into smaller parts (of interest):

```
In [110]: l[2:5]  ❶
Out[110]: [2.5, 1.0, 1.5]
```

❶ Return the third through fifth elements.

Table 3-2 provides a summary of selected operations and methods of the `list` object.

Table 3-2. Selected operations and methods of list objects

Method	Arguments	Returns/result
`l[i] = x`	`[i]`	Replaces i-th element by x
`l[i:j:k] = s`	`[i:j:k]`	Replaces every k-th element from i to $j-1$ by s
`append`	`(x)`	Appends x to object
`count`	`(x)`	Number of occurrences of object x
`del l[i:j:k]`	`[i:j:k]`	Deletes elements with index values i to $j-1$
`extend`	`(s)`	Appends all elements of s to object
`index`	`(x[, i[, j]])`	First index of x between elements i and $j-1$
`insert`	`(i, x)`	Inserts x at/before index i

`remove`	(*i*)	Removes element with index *i*
`pop`	(*i*)	Removes element with index *i* and returns it
`reverse`	()	Reverses all items in place
`sort`	([cmp[, key[, reverse]]])	Sorts all items in place

Excursion: Control Structures

Although a topic in themselves, *control structures* like `for` loops are maybe best introduced in Python based on `list` objects. This is due to the fact that looping in general takes place over `list` objects, which is quite different to what is often the standard in other languages. Take the following example. The `for` loop loops over the elements of the `list` object `l` with index values 2 to 4 and prints the square of the respective elements. Note the importance of the indentation (whitespace) in the second line:

```
In [111]: for element in l[2:5]:
              print(element ** 2)
          6.25
          1.0
          2.25
```

This provides a really high degree of flexibility in comparison to the typical counter-based looping. Counter-based looping is also an option with Python, but is accomplished using the `range` object:

```
In [112]: r = range(0, 8, 1)  ❶
          r
Out[112]: range(0, 8)

In [113]: type(r)
Out[113]: range
```

❶ Parameters are `start`, `end`, and `step-size`.

For comparison, the same loop is implemented using `range()` as follows:

```
In [114]: for i in range(2, 5):
              print(l[i] ** 2)
```

```
6.25
1.0
2.25
```

LOOPING OVER LISTS

In Python you can loop over arbitrary `list` objects, no matter what the content of the object is. This often avoids the introduction of a counter.

Python also provides the typical (conditional) control elements `if`, `elif`, and `else`. Their use is comparable in other languages:

```
In [115]: for i in range(1, 10):
              if i % 2 == 0:  ❶
                  print("%d is even" % i)
              elif i % 3 == 0:
                  print("%d is multiple of 3" % i)
              else:
                  print("%d is odd" % i)
          1 is odd
          2 is even
          3 is multiple of 3
          4 is even
          5 is odd
          6 is even
          7 is odd
          8 is even
          9 is multiple of 3
```

❶ `%` stands for modulo.

Similarly, `while` provides another means to control the flow:

```
In [116]: total = 0
          while total < 100:
              total += 1
          print(total)
          100
```

A specialty of Python is so-called *list comprehensions*. Instead of looping over existing `list` objects, this approach generates `list` objects via loops in a rather compact fashion:

```
In [117]: m = [i ** 2 for i in range(5)]
          m
Out[117]: [0, 1, 4, 9, 16]
```

In a certain sense, this already provides a first means to generate “something like” vectorized code in that loops are implicit rather than explicit (vectorization of code is discussed in more detail in Chapters 4 and 5).

Excursion: Functional Programming

Python provides a number of tools for functional programming support as well —i.e., the application of a function to a whole set of inputs (in our case `list` objects). Among these tools are `filter()`, `map()`, and `reduce()`. However, one needs a function definition first. To start with something really simple, consider a function `f()` that returns the square of the input `x`:

```
In [118]: def f(x):
              return x ** 2
          f(2)
Out[118]: 4
```

Of course, functions can be arbitrarily complex, with multiple input/parameter objects and even multiple outputs (return objects). However, consider the following function:

```
In [119]: def even(x):
              return x % 2 == 0
          even(3)
Out[119]: False
```

The return object is a Boolean. Such a function can be applied to a whole `list` object by using `map()`:

```
In [120]: list(map(even, range(10)))
Out[120]: [True, False, True, False, True, False, True, False, True, False]
```

To this end, one can also provide a function definition directly as an argument to `map()`, making use of `lambda` or *anonymous* functions:

```
In [121]: list(map(lambda x: x ** 2, range(10)))
Out[121]: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Functions can also be used to filter a `list` object. In the following example, the filter returns elements of a `list` object that match the Boolean condition as defined by the `even` function:

```
In [122]: list(filter(even, range(15)))
Out[122]: [0, 2, 4, 6, 8, 10, 12, 14]
```

> **LIST COMPREHENSIONS, FUNCTIONAL PROGRAMMING, ANONYMOUS FUNCTIONS**
>
> It can be considered good practice to avoid loops on the Python level as far as possible. List comprehensions and functional programming tools like `filter()`, `map()`, and `reduce()` provide means to write code without (explicit) loops that is both compact and in general more readable. `lambda` or anonymous functions are also powerful tools in this context.

Dicts

`dict` objects are dictionaries, and also mutable sequences, that allow data retrieval by keys (which can, for example, be `str` objects). They are so-called *key-value stores*. While `list` objects are ordered and sortable, `dict` objects are unordered and not sortable, in general.[5] An example best illustrates further differences to `list` objects. Curly braces are what define `dict` objects:

```
In [123]: d = {
               'Name' : 'Angela Merkel',
               'Country' : 'Germany',
               'Profession' : 'Chancelor',
               'Age' : 64
               }
          type(d)
Out[123]: dict

In [124]: print(d['Name'], d['Age'])
          Angela Merkel 64
```

Again, this class of objects has a number of built-in methods:

```
In [125]: d.keys()
Out[125]: dict_keys(['Name', 'Country', 'Profession', 'Age'])

In [126]: d.values()
Out[126]: dict_values(['Angela Merkel', 'Germany', 'Chancelor', 64])

In [127]: d.items()
Out[127]: dict_items([('Name', 'Angela Merkel'), ('Country', 'Germany'),
           ('Profession', 'Chancelor'), ('Age', 64)])

In [128]: birthday = True
          if birthday:
              d['Age'] += 1
          print(d['Age'])
          65
```

There are several methods to get `iterator` objects from a `dict` object. The `iterator` objects behave like `list` objects when iterated over:

```
In [129]: for item in d.items():
              print(item)
          ('Name', 'Angela Merkel')
          ('Country', 'Germany')
          ('Profession', 'Chancelor')
          ('Age', 65)

In [130]: for value in d.values():
              print(type(value))
          <class 'str'>
          <class 'str'>
          <class 'str'>
          <class 'int'>
```

Table 3-3 provides a summary of selected operations and methods of the `dict` object.

Table 3-3. Selected operations and methods of dict objects

Method	Arguments	Returns/result
`d[k]`	`[k]`	Item of `d` with key *k*
`d[k] = x`	`[k]`	Sets item key *k* to *x*

`del d[k]`	`[k]`	Deletes item with key *k*
`clear`	`()`	Removes all items
`copy`	`()`	Makes a copy
`has_key`	`(k)`	`True` if *k* is a key
`items`	`()`	Iterator over all items
`keys`	`()`	Iterator over all keys
`values`	`()`	Iterator over all values
`popitem`	`(k)`	Returns and removes item with key *k*
`update`	`([e])`	Updates items with items from *e*

Sets

The final data structure this section covers is the `set` object. Although set theory is a cornerstone of mathematics and also of financial theory, there are not too many practical applications for `set` objects. The objects are unordered collections of other objects, containing every element only once:

```
In [131]: s = set(['u', 'd', 'ud', 'du', 'd', 'du'])
          s
Out[131]: {'d', 'du', 'u', 'ud'}

In [132]: t = set(['d', 'dd', 'uu', 'u'])
```

With `set` objects, one can implement basic operations on sets as in mathematical set theory. For example, one can generate unions, intersections, and differences:

```
In [133]: s.union(t)  ❶
Out[133]: {'d', 'dd', 'du', 'u', 'ud', 'uu'}

In [134]: s.intersection(t)  ❷
Out[134]: {'d', 'u'}

In [135]: s.difference(t)  ❸
Out[135]: {'du', 'ud'}

In [136]: t.difference(s)  ❹
```

```
Out[136]: {'dd', 'uu'}

In [137]: s.symmetric_difference(t)  ❺
Out[137]: {'dd', 'du', 'ud', 'uu'}
```

❶ All of `s` and `t`.
❷ Items in both `s` and `t`.
❸ Items in `s` but not in `t`.
❹ Items in `t` but not in `s`.
❺ Items in either `s` or `t` but not both.

One application of `set` objects is to get rid of duplicates in a `list` object:

```
In [138]: from random import randint
          l = [randint(0, 10) for i in range(1000)]  ❶
          len(l)  ❷
Out[138]: 1000

In [139]: l[:20]
Out[139]: [4, 2, 10, 2, 1, 10, 0, 6, 0, 8, 10, 9, 2, 4, 7, 8, 10, 8, 8, 2]

In [140]: s = set(l)
          s
Out[140]: {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
```

❶ 1,000 random integers between 0 and 10.
❷ Number of elements in `l`.

Conclusion

The basic Python interpreter provides a rich set of flexible data structures. From a finance point of view, the following can be considered the most important ones:

Basic data types

> In Python in general and finance in particular, the classes `int`, `float`, `bool`, and `str` provide the atomic data types.

Standard data structures

The classes `tuple`, `list`, `dict`, and `set` have many application areas in finance, with `list` being a flexible all-rounder for a number use cases.

Further Resources

With regard to data types and structures, this chapter focuses on those topics that might be of particular importance for financial algorithms and applications. However, it can only represent a starting point for the exploration of data structures and data modeling in Python.

There are a number of valuable resources available to go deeper from here. The official documentation for Python data structures is found at *https://docs.python.org/3/tutorial/datastructures.html*.

Good references in book form are:

- Goodrich, Michael, et al. (2013). *Data Structures and Algorithms in Python*. Hoboken, NJ: John Wiley & Sons.
- Harrison, Matt (2017). *Illustrated Guide to Python 3*. CreateSpace Treading on Python Series.
- Ramalho, Luciano (2016). *Fluent Python*. Sebastopol, CA: O'Reilly.

For an introduction to regular expressions, see:

- Fitzgerald, Michael (2012). *Introducing Regular Expressions*. Sebastopol, CA: O'Reilly.

---

1 The `Cython` package brings static typing and compiling features to Python that are comparable to those in C. In fact, `Cython` is not only a *package*, it is a full-fledged hybrid *programming language* combining Python and C.

2 This is different in Python 2.x, where floor division is the default. Floor division in Python 3.x is accomplished by `3 // 4`, which gives `0` as the result.

3 Here and in the following discussion, terms like `float`, `float` object, etc. are used interchangeably, acknowledging that every `float` is also an *object*. The same holds true for other object types.

4 It is not possible to go into detail here, but there is a wealth of information available on the internet about regular expressions in general and for Python in particular. For an introduction to this topic, refer to Fitzgerald (2012).

5 There are variants to the standard `dict` object, including among others an `OrderedDict` subclass, which remembers the order in which entries are added. See *https://docs.python.org/3/library/collections.html*.

Chapter 4. Numerical Computing with NumPy

Computers are useless. They can only give answers.

—Pablo Picasso

Although the Python interpreter itself already brings a rich variety of data structures with it, `NumPy` and other libraries add to these in a valuable fashion. This chapter focuses on `NumPy`, which provides a multidimensional array object to store homogeneous or heterogeneous data arrays and supports vectorization of code.

The chapter covers the following data structures:

Object type	Meaning	Used for
`ndarray` (regular)	*n*-dimensional array object	Large arrays of numerical data
`ndarray` (record)	2-dimensional array object	Tabular data organized in columns

This chapter is organized as follows:

"Arrays of Data"

This section is about the handling of arrays of data with pure Python code.

"Regular NumPy Arrays"

This is the core section about the regular `NumPy` `ndarray` class, the workhorse in almost all data-intensive Python use cases involving numerical data.

"Structured NumPy Arrays"

This brief section introduces structured (or *record*) `ndarray` objects for the handling of tabular data with columns.

"Vectorization of Code"

In this section, vectorization of code is discussed along with its benefits; the section also discusses the importance of memory layout in certain scenarios.

Arrays of Data

The previous chapter showed that Python provides some quite useful and flexible general data structures. In particular, `list` objects can be considered a real workhorse with many convenient characteristics and application areas. Using such a flexible (mutable) data structure has a cost, in the form of relatively high memory usage, slower performance, or both. However, scientific and financial applications generally have a need for high-performing operations on special data structures. One of the most important data structures in this regard is the *array*. Arrays generally structure other (fundamental) objects of the *same data type* in rows and columns.

Assume for the moment that only numbers are relevant, although the concept generalizes to other types of data as well. In the simplest case, a one-dimensional array then represents, mathematically speaking, a *vector* of, in general, real numbers, internally represented by `float` objects. It then consists of a *single* row or column of elements only. In the more common case, an array represents an $i \times j$ *matrix* of elements. This concept generalizes to $i \times j \times k$ *cubes* of elements in three dimensions as well as to general *n*-dimensional arrays of shape $i \times j \times k \times l \times \ldots$.

Mathematical disciplines like linear algebra and vector space theory illustrate that such mathematical structures are of high importance in a number of scientific disciplines and fields. It can therefore prove fruitful to have available a specialized class of data structures explicitly designed to handle arrays conveniently and efficiently. This is where the Python library `NumPy` comes into play, with its powerful `ndarray` class. Before introducing this class in the next section, this section illustrates two alternatives for the handling of arrays.

Arrays with Python Lists

Arrays can be constructed with the built-in data structures presented in the previous chapter. `list` objects are particularly suited to accomplishing this task.

A simple `list` can already be considered a one-dimensional array:

```
In [1]: v = [0.5, 0.75, 1.0, 1.5, 2.0]  ❶
```

❶ `list` object with numbers.

Since `list` objects can contain arbitrary other objects, they can also contain other `list` objects. In that way, two- and higher-dimensional arrays are easily constructed by nested `list` objects:

```
In [2]: m = [v, v, v]  ❶
        m  ❷
Out[2]: [[0.5, 0.75, 1.0, 1.5, 2.0],
         [0.5, 0.75, 1.0, 1.5, 2.0],
         [0.5, 0.75, 1.0, 1.5, 2.0]]
```

❶ `list` object with `list` objects …

❷ … resulting in a matrix of numbers.

One can also easily select rows via simple indexing or single elements via double indexing (whole columns, however, are not so easy to select):

```
In [3]: m[1]
Out[3]: [0.5, 0.75, 1.0, 1.5, 2.0]

In [4]: m[1][0]
Out[4]: 0.5
```

Nesting can be pushed further for even more general structures:

```
In [5]: v1 = [0.5, 1.5]
        v2 = [1, 2]
        m = [v1, v2]
        c = [m, m]  ❶
        c
Out[5]: [[[0.5, 1.5], [1, 2]], [[0.5, 1.5], [1, 2]]]

In [6]: c[1][1][0]
Out[6]: 1
```

❶ Cube of numbers.

Note that combining objects in the way just presented generally works with

reference pointers to the original objects. What does that mean in practice? Have a look at the following operations:

```
In [7]: v = [0.5, 0.75, 1.0, 1.5, 2.0]
        m = [v, v, v]
        m
Out[7]: [[0.5, 0.75, 1.0, 1.5, 2.0],
         [0.5, 0.75, 1.0, 1.5, 2.0],
         [0.5, 0.75, 1.0, 1.5, 2.0]]
```

Now change the value of the first element of the `v` object and see what happens to the `m` object:

```
In [8]: v[0] = 'Python'
        m
Out[8]: [['Python', 0.75, 1.0, 1.5, 2.0],
         ['Python', 0.75, 1.0, 1.5, 2.0],
         ['Python', 0.75, 1.0, 1.5, 2.0]]
```

This can be avoided by using the `deepcopy()` function of the `copy` module:

```
In [9]: from copy import deepcopy
        v = [0.5, 0.75, 1.0, 1.5, 2.0]
        m = 3 * [deepcopy(v), ]  ❶
        m
Out[9]: [[0.5, 0.75, 1.0, 1.5, 2.0],
         [0.5, 0.75, 1.0, 1.5, 2.0],
         [0.5, 0.75, 1.0, 1.5, 2.0]]

In [10]: v[0] = 'Python'  ❷
         m  ❸
Out[10]: [[0.5, 0.75, 1.0, 1.5, 2.0],
          [0.5, 0.75, 1.0, 1.5, 2.0],
          [0.5, 0.75, 1.0, 1.5, 2.0]]
```

❶ Instead of reference pointer, physical copies are used.
❷ As a consequence, a change in the original object …
❸ … does not have any impact anymore.

The Python array Class

There is a dedicated `array` module available in Python. According to the

documentation:

> *This module defines an object type which can compactly represent an array of basic values: characters, integers, floating point numbers. Arrays are sequence types and behave very much like lists, except that the type of objects stored in them is constrained. The type is specified at object creation time by using a type code, which is a single character.*

Consider the following code, which instantiates an `array` object out of a `list` object:

```
In [11]: v = [0.5, 0.75, 1.0, 1.5, 2.0]

In [12]: import array

In [13]: a = array.array('f', v)  ❶
         a
Out[13]: array('f', [0.5, 0.75, 1.0, 1.5, 2.0])

In [14]: a.append(0.5)  ❷
         a
Out[14]: array('f', [0.5, 0.75, 1.0, 1.5, 2.0, 0.5])

In [15]: a.extend([5.0, 6.75])  ❷
         a
Out[15]: array('f', [0.5, 0.75, 1.0, 1.5, 2.0, 0.5, 5.0, 6.75])

In [16]: 2 * a  ❸
Out[16]: array('f', [0.5, 0.75, 1.0, 1.5, 2.0, 0.5, 5.0, 6.75, 0.5, 0.75, 1.0,
          1.5, 2.0, 0.5, 5.0, 6.75])
```

❶ The instantiation of the `array` object with `float` as the type code.

❷ Major methods work similar to those of the `list` object.

❸ Although “scalar multiplication” works in principle, the result is not the mathematically expected one; rather, the elements are repeated.

Trying to append an object of a different data type than the one specified raises a `TypeError`:

```
In [17]: a.append('string')  ❶

         ---------------------------------------
         TypeErrorTraceback (most recent call last)
         <ipython-input-17-14cd6281866b> in <module>()
```

```
----> 1 a.append('string')  ❶

TypeError: must be real number, not str

In [18]: a.tolist()  ❷
Out[18]: [0.5, 0.75, 1.0, 1.5, 2.0, 0.5, 5.0, 6.75]
```

❶ Only `float` objects can be appended; other data types/type codes raise errors.

❷ However, the `array` object can easily be converted back to a `list` object if such flexibility is required.

An advantage of the `array` class is that it has built-in storage and retrieval functionality:

```
In [19]: f = open('array.apy', 'wb')  ❶
         a.tofile(f)  ❷
         f.close()  ❸

In [20]: with open('array.apy', 'wb') as f:  ❹
             a.tofile(f)  ❹

In [21]: !ls -n arr*  ❺
         -rw-r--r--@ 1 503  20  32 Nov  7 11:46 array.apy
```

❶ Opens a file on disk for writing binary data.

❷ Writes the `array` data to the file.

❸ Closes the file.

❹ Alternative: uses a `with` context for the same operation.

❺ Shows the file as written on disk.

As before, the data type of the `array` object is of importance when reading the data from disk:

```
In [22]: b = array.array('f')  ❶

In [23]: with open('array.apy', 'rb') as f:  ❷
             b.fromfile(f, 5)  ❸

In [24]: b  ❹
Out[24]: array('f', [0.5, 0.75, 1.0, 1.5, 2.0])

In [25]: b = array.array('d')  ❹
```

```
In [26]: with open('array.apy', 'rb') as f:
             b.fromfile(f, 2)  ❺

In [27]: b  ❻
Out[27]: array('d', [0.0004882813645963324, 0.12500002956949174])
```

❶ Instantiates a new `array` object with type code `float`.
❷ Opens the file for reading binary data …
❸ … and reads five elements in the `b` object.
❹ Instantiates a new `array` object with type code `double`.
❺ Reads two elements from the file.
❻ The difference in type codes leads to “wrong” numbers.

Regular NumPy Arrays

Composing array structures with `list` objects works, somewhat. But it is not really convenient, and the `list` class has not been built with this specific goal in mind. It has rather a much broader and more general scope. The `array` class is a bit more specialized, providing some useful features for working with arrays of data. However, a truly specialized class could be really beneficial to handle array-type structures.

The Basics

`numpy.ndarray` is just such a class, built with the specific goal of handling *n*-dimensional arrays both conveniently and efficiently—i.e., in a highly performant manner. The basic handling of instances of this class is again best illustrated by examples:

```
In [28]: import numpy as np  ❶

In [29]: a = np.array([0, 0.5, 1.0, 1.5, 2.0])  ❷
         a
Out[29]: array([0. , 0.5, 1. , 1.5, 2. ])

In [30]: type(a)  ❷
Out[30]: numpy.ndarray
```

```
In [31]: a = np.array(['a', 'b', 'c'])  ❸
         a
Out[31]: array(['a', 'b', 'c'], dtype='<U1')

In [32]: a = np.arange(2, 20, 2)  ❹
         a
Out[32]: array([ 2,  4,  6,  8, 10, 12, 14, 16, 18])

In [33]: a = np.arange(8, dtype=np.float)  ❺
         a
Out[33]: array([0., 1., 2., 3., 4., 5., 6., 7.])

In [34]: a[5:]  ❻
Out[34]: array([5., 6., 7.])

In [35]: a[:2]  ❻
Out[35]: array([0., 1.])
```

❶ Imports the `numpy` package.
❷ Creates an `ndarray` object out of a `list` object with `floats`.
❸ Creates an `ndarray` object out of a `list` object with `strs`.
❹ `np.arange()` works similar to `range()` …
❺ … but takes as additional input the `dtype` parameter.
❻ With one-dimensional `ndarray` objects, indexing works as usual.

A major feature of the `ndarray` class is the *multitude of built-in methods*. For instance:

```
In [36]: a.sum()  ❶
Out[36]: 28.0

In [37]: a.std()  ❷
Out[37]: 2.29128784747792

In [38]: a.cumsum()  ❸
Out[38]: array([ 0.,  1.,  3.,  6., 10., 15., 21., 28.])
```

❶ The sum of all elements.
❷ The standard deviation of the elements.
❸ The cumulative sum of all elements (starting at index position 0).

Another major feature is the (vectorized) *mathematical operations* defined on `ndarray` objects:

```
In [39]: l = [0., 0.5, 1.5, 3., 5.]
         2 * l  ❶
Out[39]: [0.0, 0.5, 1.5, 3.0, 5.0, 0.0, 0.5, 1.5, 3.0, 5.0]

In [40]: a
Out[40]: array([0., 1., 2., 3., 4., 5., 6., 7.])

In [41]: 2 * a  ❷
Out[41]: array([ 0.,  2.,  4.,  6.,  8., 10., 12., 14.])

In [42]: a ** 2  ❸
Out[42]: array([ 0.,  1.,  4.,  9., 16., 25., 36., 49.])

In [43]: 2 ** a  ❹
Out[43]: array([  1.,   2.,   4.,   8.,  16.,  32.,  64., 128.])

In [44]: a ** a  ❺
Out[44]: array([1.00000e+00, 1.00000e+00, 4.00000e+00, 2.70000e+01, 2.56000e+02,
                3.12500e+03, 4.66560e+04, 8.23543e+05])
```

❶ Scalar multiplication with `list` objects leads to a repetition of elements.

❷ By contrast, working with `ndarray` objects implements a proper scalar multiplication.

❸ This calculates element-wise the square values.

❹ This interprets the elements of the `ndarray` as the powers.

❺ This calculates the power of every element to itself.

Universal functions are another important feature of the `NumPy` package. They are "universal" in the sense that they in general operate on `ndarray` objects as well as on basic Python data types. However, when applying universal functions to, say, a Python `float` object, one needs to be aware of the reduced performance compared to the same functionality found in the `math` module:

```
In [45]: np.exp(a)  ❶
Out[45]: array([1.00000000e+00, 2.71828183e+00, 7.38905610e+00, 2.00855369e+01,
                5.45981500e+01, 1.48413159e+02, 4.03428793e+02, 1.09663316e+03])

In [46]: np.sqrt(a)  ❷
Out[46]: array([0.        , 1.        , 1.41421356, 1.73205081, 2.        ,
                2.23606798, 2.44948974, 2.64575131])

In [47]: np.sqrt(2.5)  ❸
Out[47]: 1.5811388300841898
```

```
In [48]: import math  ❹

In [49]: math.sqrt(2.5)  ❹
Out[49]: 1.5811388300841898

In [50]: math.sqrt(a)  ❺

         ---------------------------------------
         TypeErrorTraceback (most recent call last)
         <ipython-input-50-b39de4150838> in <module>()
         ----> 1 math.sqrt(a)  ❺

         TypeError: only size-1 arrays can be converted to Python scalars

In [51]: %timeit np.sqrt(2.5)  ❻
         722 ns ± 13.7 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops
          each)

In [52]: %timeit math.sqrt(2.5)  ❼
         91.8 ns ± 4.13 ns per loop (mean ± std. dev. of 7 runs, 10000000 loops
          each)
```

❶ Calculates the exponential values element-wise.
❷ Calculates the square root for every element.
❸ Calculates the square root for a Python `float` object.
❹ The same calculation, this time using the `math` module.
❺ The `math.sqrt()` function cannot be applied to the `ndarray` object directly.
❻ Applying the universal function `np.sqrt()` to a Python `float` object …
❼ … is much slower than the same operation with the `math.sqrt()` function.

Multiple Dimensions

The transition to more than one dimension is seamless, and all features presented so far carry over to the more general cases. In particular, the indexing system is made consistent across all dimensions:

```
In [53]: b = np.array([a, a * 2])  ❶
         b
Out[53]: array([[ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.],
                [ 0.,  2.,  4.,  6.,  8., 10., 12., 14.]])

In [54]: b[0]  ❷
Out[54]: array([0., 1., 2., 3., 4., 5., 6., 7.])
```

```
In [55]: b[0, 2]  ❸
Out[55]: 2.0

In [56]: b[:, 1]  ❹
Out[56]: array([1., 2.])

In [57]: b.sum()  ❺
Out[57]: 84.0

In [58]: b.sum(axis=0)  ❻
Out[58]: array([ 0.,  3.,  6.,  9., 12., 15., 18., 21.])

In [59]: b.sum(axis=1)  ❼
Out[59]: array([28., 56.])
```

❶ Constructs a two-dimensional `ndarray` object out of the one-dimensional one.
❷ Selects the first row.
❸ Selects the third element in the first row; indices are separated, within the brackets, by a comma.
❹ Selects the second column.
❺ Calculates the sum of *all* values.
❻ Calculates the sum along the first axis; i.e., column-wise.
❼ Calculates the sum along the second axis; i.e., row-wise.

There are a number of ways to initialize (instantiate) `ndarray` objects. One is as presented before, via `np.array`. However, this assumes that all elements of the array are already available. In contrast, one might like to have the `ndarray` objects instantiated first to populate them later with results generated during the execution of code. To this end, one can use the following functions:

```
In [60]: c = np.zeros((2, 3), dtype='i', order='C')  ❶
         c
Out[60]: array([[0, 0, 0],
                [0, 0, 0]], dtype=int32)

In [61]: c = np.ones((2, 3, 4), dtype='i', order='C')  ❷
         c
Out[61]: array([[[1, 1, 1, 1],
                 [1, 1, 1, 1],
                 [1, 1, 1, 1]],
```

```
                [[1, 1, 1, 1],
                 [1, 1, 1, 1],
                 [1, 1, 1, 1]]], dtype=int32)

In [62]: d = np.zeros_like(c, dtype='f16', order='C')  ❸
         d
Out[62]: array([[[0., 0., 0., 0.],
                 [0., 0., 0., 0.],
                 [0., 0., 0., 0.]],

                [[0., 0., 0., 0.],
                 [0., 0., 0., 0.],
                 [0., 0., 0., 0.]]], dtype=float128)

In [63]: d = np.ones_like(c, dtype='f16', order='C')  ❸
         d
Out[63]: array([[[1., 1., 1., 1.],
                 [1., 1., 1., 1.],
                 [1., 1., 1., 1.]],

                [[1., 1., 1., 1.],
                 [1., 1., 1., 1.],
                 [1., 1., 1., 1.]]], dtype=float128)

In [64]: e = np.empty((2, 3, 2))  ❹
         e
Out[64]: array([[[0.00000000e+000, 0.00000000e+000],
                 [0.00000000e+000, 0.00000000e+000],
                 [0.00000000e+000, 0.00000000e+000]],

                [[0.00000000e+000, 0.00000000e+000],
                 [0.00000000e+000, 7.49874326e+247],
                 [1.28822975e-231, 4.33190018e-311]]])

In [65]: f = np.empty_like(c)  ❹
         f
Out[65]: array([[[         0,          0,          0,          0],
                 [         0,          0,          0,          0],
                 [         0,          0,          0,          0]],

                [[         0,          0,          0,          0],
                 [         0,          0,  740455269, 1936028450],
                 [         0,  268435456, 1835316017,       2041]]], dtype=int32)

In [66]: np.eye(5)  ❺
Out[66]: array([[1., 0., 0., 0., 0.],
                [0., 1., 0., 0., 0.],
                [0., 0., 1., 0., 0.],
                [0., 0., 0., 1., 0.],
```

```
                [0., 0., 0., 0., 1.]])

In [67]: g = np.linspace(5, 15, 12) ❻
         g
Out[67]: array([ 5.        ,  5.90909091,  6.81818182,  7.72727273,  8.63636364,
                 9.54545455, 10.45454545, 11.36363636, 12.27272727, 13.18181818,
                14.09090909, 15.        ])
```

❶ Creates an `ndarray` object prepopulated with zeros.

❷ Creates an `ndarray` object prepopulated with ones.

❸ The same, but takes another `ndarray` object to infer the shape.

❹ Creates an `ndarray` object not prepopulated with anything (numbers depend on the bits present in the memory).

❺ Creates a square matrix as an `ndarray` object with the diagonal populated by ones.

❻ Creates a one-dimensional `ndarray` object with evenly spaced intervals between numbers; parameters used are `start`, `end`, and `num` (number of elements).

For all these functions, one can provide the following parameters:

`shape`

Either an `int`, a sequence of `int` objects, or a reference to another `ndarray`

`dtype` (optional)

A `dtype`—these are `NumPy`-specific data types for `ndarray` objects

`order` (optional)

The order in which to store elements in memory: `C` for C-like (i.e., row-wise) or `F` for Fortran-like (i.e., column-wise)

Here, it becomes obvious how `NumPy` specializes the construction of arrays with the `ndarray` class, in comparison to the `list` -based approach:

- The `ndarray` object has built-in *dimensions* (axes).
- The `ndarray` object is *immutable*; its length (size) is fixed.
- It only allows for a *single data type* (`np.dtype`) for the whole array.

The `array` class by contrast shares only the characteristic of allowing for a single data type (type code, `dtype`).

The role of the `order` parameter is discussed later in the chapter. Table 4-1 provides an overview of selected `np.dtype` objects (i.e., the basic data types `NumPy` allows).

Table 4-1. NumPy dtype objects

dtype	Description	Example
`?`	Boolean	`?` (`True` or `False`)
`i`	Signed integer	`i8` (64-bit)
`u`	Unsigned integer	`u8` (64-bit)
`f`	Floating point	`f8` (64-bit)
`c`	Complex floating point	`c32` (256-bit)
`m`	`timedelta`	`m` (64-bit)
`M`	`datetime`	`M` (64-bit)
`O`	Object	`O` (pointer to object)
`U`	Unicode	`U24` (24 Unicode characters)
`V`	Raw data (void)	`V12` (12-byte data block)

Metainformation

Every `ndarray` object provides access to a number of useful attributes:

```
In [68]: g.size  ❶
Out[68]: 12

In [69]: g.itemsize  ❷
Out[69]: 8

In [70]: g.ndim  ❸
Out[70]: 1
```

```
In [71]: g.shape  ❹
Out[71]: (12,)

In [72]: g.dtype  ❺
Out[72]: dtype('float64')

In [73]: g.nbytes  ❻
Out[73]: 96
```

❶ The number of elements.
❷ The number of bytes used to represent one element.
❸ The number of dimensions.
❹ The shape of the `ndarray` object.
❺ The `dtype` of the elements.
❻ The total number of bytes used in memory.

Reshaping and Resizing

Although `ndarray` objects are immutable by default, there are multiple options to reshape and resize such an object. While *reshaping* in general just provides another *view* on the same data, *resizing* in general creates a *new* (temporary) object. First, some examples of reshaping:

```
In [74]: g = np.arange(15)

In [75]: g
Out[75]: array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14])

In [76]: g.shape  ❶
Out[76]: (15,)

In [77]: np.shape(g) ❶
Out[77]: (15,)

In [78]: g.reshape((3, 5))  ❷
Out[78]: array([[ 0,  1,  2,  3,  4],
                [ 5,  6,  7,  8,  9],
                [10, 11, 12, 13, 14]])

In [79]: h = g.reshape((5, 3))  ❸
         h
Out[79]: array([[ 0,  1,  2],
                [ 3,  4,  5],
```

```
                [ 6,  7,  8],
                [ 9, 10, 11],
                [12, 13, 14]])

In [80]: h.T  ❹
Out[80]: array([[ 0,  3,  6,  9, 12],
                [ 1,  4,  7, 10, 13],
                [ 2,  5,  8, 11, 14]])

In [81]: h.transpose()  ❹
Out[81]: array([[ 0,  3,  6,  9, 12],
                [ 1,  4,  7, 10, 13],
                [ 2,  5,  8, 11, 14]])
```

❶ The shape of the original `ndarray` object.
❷ Reshaping to two dimensions (memory view).
❸ Creating a new object.
❹ The transpose of the new `ndarray` object.

During a reshaping operation, the total number of elements in the `ndarray` object is unchanged. During a resizing operation, this number changes—it either decreases ("down-sizing") or increases ("up-sizing"). Here some examples of resizing:

```
In [82]: g
Out[82]: array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14])

In [83]: np.resize(g, (3, 1))  ❶
Out[83]: array([[0],
                [1],
                [2]])

In [84]: np.resize(g, (1, 5))  ❶
Out[84]: array([[0, 1, 2, 3, 4]])

In [85]: np.resize(g, (2, 5))  ❶
Out[85]: array([[0, 1, 2, 3, 4],
                [5, 6, 7, 8, 9]])

In [86]: n = np.resize(g, (5, 4))  ❷
         n
Out[86]: array([[ 0,  1,  2,  3],
                [ 4,  5,  6,  7],
                [ 8,  9, 10, 11],
                [12, 13, 14,  0],
                [ 1,  2,  3,  4]])
```

❶ Two dimensions, down-sizing.
❷ Two dimensions, up-sizing.

Stacking is a special operation that allows the horizontal or vertical combination of two `ndarray` objects. However, the size of the "connecting" dimension must be the same:

```
In [87]: h
Out[87]: array([[ 0,  1,  2],
                [ 3,  4,  5],
                [ 6,  7,  8],
                [ 9, 10, 11],
                [12, 13, 14]])

In [88]: np.hstack((h, 2 * h))  ❶
Out[88]: array([[ 0,  1,  2,  0,  2,  4],
                [ 3,  4,  5,  6,  8, 10],
                [ 6,  7,  8, 12, 14, 16],
                [ 9, 10, 11, 18, 20, 22],
                [12, 13, 14, 24, 26, 28]])

In [89]: np.vstack((h, 0.5 * h))  ❷
Out[89]: array([[ 0. ,  1. ,  2. ],
                [ 3. ,  4. ,  5. ],
                [ 6. ,  7. ,  8. ],
                [ 9. , 10. , 11. ],
                [12. , 13. , 14. ],
                [ 0. ,  0.5,  1. ],
                [ 1.5,  2. ,  2.5],
                [ 3. ,  3.5,  4. ],
                [ 4.5,  5. ,  5.5],
                [ 6. ,  6.5,  7. ]])
```

❶ Horizontal stacking of two `ndarray` objects.
❷ Vertical stacking of two `ndarray` objects.

Another special operation is the *flattening* of a multidimensional `ndarray` object to a one-dimensional one. One can choose whether the flattening happens row-by-row (`C` order) or column-by-column (`F` order):

```
In [90]: h
Out[90]: array([[ 0,  1,  2],
                [ 3,  4,  5],
                [ 6,  7,  8],
```

```
                [ 9, 10, 11],
                [12, 13, 14]])

In [91]: h.flatten()  ❶
Out[91]: array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14])

In [92]: h.flatten(order='C')  ❶
Out[92]: array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14])

In [93]: h.flatten(order='F')  ❷
Out[93]: array([ 0,  3,  6,  9, 12,  1,  4,  7, 10, 13,  2,  5,  8, 11, 14])

In [94]: for i in h.flat:  ❸
             print(i, end=',')
         0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,
In [95]: for i in h.ravel(order='C'):  ❹
             print(i, end=',')
         0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,
In [96]: for i in h.ravel(order='F'):  ❹
             print(i, end=',')
         0,3,6,9,12,1,4,7,10,13,2,5,8,11,14,
```

❶ The default order for flattening is `C`.

❷ Flattening with `F` order.

❸ The `flat` attribute provides a flat iterator (`C` order).

❹ The `ravel()` method is an alternative to `flatten()`.

Boolean Arrays

Comparison and logical operations in general work on `ndarray` objects the same way, element-wise, as on standard Python data types. Evaluating conditions yield by default a Boolean `ndarray` object (`dtype` is `bool`):

```
In [97]: h
Out[97]: array([[ 0,  1,  2],
                [ 3,  4,  5],
                [ 6,  7,  8],
                [ 9, 10, 11],
                [12, 13, 14]])

In [98]: h > 8  ❶
Out[98]: array([[False, False, False],
                [False, False, False],
                [False, False, False],
```

```
                [ True,  True,  True],
                [ True,  True,  True]])

In [99]: h <= 7  ❷
Out[99]: array([[ True,  True,  True],
                [ True,  True,  True],
                [ True,  True, False],
                [False, False, False],
                [False, False, False]])

In [100]: h == 5  ❸
Out[100]: array([[False, False, False],
                 [False, False,  True],
                 [False, False, False],
                 [False, False, False],
                 [False, False, False]])

In [101]: (h == 5).astype(int)  ❹
Out[101]: array([[0, 0, 0],
                 [0, 0, 1],
                 [0, 0, 0],
                 [0, 0, 0],
                 [0, 0, 0]])

In [102]: (h > 4) & (h <= 12)  ❺
Out[102]: array([[False, False, False],
                 [False, False,  True],
                 [ True,  True,  True],
                 [ True,  True,  True],
                 [ True, False, False]])
```

❶ Is value greater than …?

❷ Is value smaller or equal than …?

❸ Is value equal to …?

❹ Present `True` and `False` as integer values 0 and 1.

❺ Is value greater than … and smaller than or equal to …?

Such Boolean arrays can be used for indexing and data selection. Notice that the following operations flatten the data:

```
In [103]: h[h > 8]  ❶
Out[103]: array([ 9, 10, 11, 12, 13, 14])

In [104]: h[(h > 4) & (h <= 12)]  ❷
Out[104]: array([ 5,  6,  7,  8,  9, 10, 11, 12])
```

```
In [105]: h[(h < 4) | (h >= 12)]  ❸
Out[105]: array([ 0,  1,  2,  3, 12, 13, 14])
```

❶ Give me all values greater than …
❷ Give me all values greater than … *and* smaller than or equal to …
❸ Give me all values greater than … *or* smaller than or equal to …

A powerful tool in this regard is the `np.where()` function, which allows the definition of actions/operations depending on whether a condition is `True` or `False`. The result of applying `np.where()` is a new `ndarray` object of the same shape as the original one:

```
In [106]: np.where(h > 7, 1, 0)  ❶
Out[106]: array([[0, 0, 0],
                 [0, 0, 0],
                 [0, 0, 1],
                 [1, 1, 1],
                 [1, 1, 1]])

In [107]: np.where(h % 2 == 0, 'even', 'odd')  ❷
Out[107]: array([['even', 'odd', 'even'],
                 ['odd', 'even', 'odd'],
                 ['even', 'odd', 'even'],
                 ['odd', 'even', 'odd'],
                 ['even', 'odd', 'even']], dtype='<U4')

In [108]: np.where(h <= 7, h * 2, h / 2)  ❸
Out[108]: array([[ 0. ,  2. ,  4. ],
                 [ 6. ,  8. , 10. ],
                 [12. , 14. ,  4. ],
                 [ 4.5,  5. ,  5.5],
                 [ 6. ,  6.5,  7. ]])
```

❶ In the new object, set `1` if `True` and `0` otherwise.
❷ In the new object, set `even` if `True` and `odd` otherwise.
❸ In the new object, set two times the `h` element if `True` and half the `h` element otherwise.

Later chapters provide more examples of these important operations on `ndarray` objects.

Speed Comparison

We'll move on to structured arrays with NumPy shortly, but let us stick with regular arrays for a moment and see what the specialization brings in terms of performance.

As a simple example, consider the generation of a matrix/array of shape 5,000 × 5,000 elements, populated with pseudo-random, standard normally distributed numbers. The sum of all elements shall then be calculated. First, the pure Python approach, where `list` comprehensions are used:

```
In [109]: import random
          I = 5000

In [110]: %time mat = [[random.gauss(0, 1) for j in range(I)] \
                       for i in range(I)]  ❶
          CPU times: user 17.1 s, sys: 361 ms, total: 17.4 s
          Wall time: 17.4 s

In [111]: mat[0][:5]  ❷
Out[111]: [-0.40594967782329183,
           -1.357757478015285,
           0.05129566894355976,
           -0.8958429976582192,
           0.6234174778878331]

In [112]: %time sum([sum(l) for l in mat])  ❸
          CPU times: user 142 ms, sys: 1.69 ms, total: 144 ms
          Wall time: 143 ms

Out[112]: -3561.944965714259

In [113]: import sys
          sum([sys.getsizeof(l) for l in mat])  ❹
Out[113]: 215200000
```

❶ The creation of the matrix via a nested `list` comprehension.

❷ Some selected random numbers from those drawn.

❸ The sums of the single `list` objects are first calculated during a list comprehension; then the sum of the sums is taken.

❹ This adds up the memory usage of all `list` objects.

Let us now turn to NumPy and see how the same problem is solved there. For convenience, the NumPy subpackage `random` offers a multitude of functions to instantiate an `ndarray` object and populate it at the same time with pseudo-

random numbers:

```
In [114]: %time mat = np.random.standard_normal((I, I))  ❶
          CPU times: user 1.01 s, sys: 200 ms, total: 1.21 s
          Wall time: 1.21 s

In [115]: %time mat.sum()  ❷
          CPU times: user 29.7 ms, sys: 1.15 ms, total: 30.8 ms
          Wall time: 29.4 ms

Out[115]: -186.12767026606448

In [116]: mat.nbytes  ❸
Out[116]: 200000000

In [117]: sys.getsizeof(mat)  ❸
Out[117]: 200000112
```

❶ Creates the `ndarray` object with standard normally distributed random numbers; it is faster by a factor of about 14.

❷ Calculates the sum of all values in the `ndarray` object; it is faster by a factor of 4.5.

❸ The `NumPy` approach also saves some memory since the memory overhead of the `ndarray` object is tiny compared to the size of the data itself.

> **USING NUMPY ARRAYS**
>
> The use of `NumPy` for array-based operations and algorithms generally results in compact, easily readable code and significant performance improvements over pure Python code.

Structured NumPy Arrays

The specialization of the `ndarray` class obviously brings a number of valuable benefits with it. However, a too narrow specialization might turn out to be too large a burden to carry for the majority of array-based algorithms and applications. Therefore, `NumPy` provides *structured* `ndarray` and *record* `recarray` objects that allow you to have a different `dtype` *per column*. What does “per column” mean? Consider the following initialization of a structured

`ndarray` object:

```
In [118]: dt = np.dtype([('Name', 'S10'), ('Age', 'i4'),
                         ('Height', 'f'), ('Children/Pets', 'i4', 2)])  ❶

In [119]: dt  ❶
Out[119]: dtype([('Name', 'S10'), ('Age', '<i4'), ('Height', '<f4'),
           ('Children/Pets', '<i4', (2,))])

In [120]: dt = np.dtype({'names': ['Name', 'Age', 'Height', 'Children/Pets'],
                         'formats':'O int float int,int'.split()})  ❷

In [121]: dt  ❷
Out[121]: dtype([('Name', 'O'), ('Age', '<i8'), ('Height', '<f8'),
           ('Children/Pets', [('f0', '<i8'), ('f1', '<i8')])])

In [122]: s = np.array([('Smith', 45, 1.83, (0, 1)),
                        ('Jones', 53, 1.72, (2, 2))], dtype=dt)  ❸

In [123]: s  ❸
Out[123]: array([('Smith', 45, 1.83, (0, 1)), ('Jones', 53, 1.72, (2, 2))],
          dtype=[('Name', 'O'), ('Age', '<i8'), ('Height', '<f8'),
           ('Children/Pets', [('f0', '<i8'), ('f1', '<i8')])])

In [124]: type(s)  ❹
Out[124]: numpy.ndarray
```

❶ The complex `dtype` is composed.

❷ An alternative syntax to achieve the same result.

❸ The structured `ndarray` is instantiated with two records.

❹ The object type is still `ndarray`.

In a sense, this construction comes quite close to the operation for initializing tables in a SQL database: one has column names and column data types, with maybe some additional information (e.g., maximum number of characters per `str` object). The single columns can now be easily accessed by their names and the rows by their index values:

```
In [125]: s['Name']  ❶
Out[125]: array(['Smith', 'Jones'], dtype=object)

In [126]: s['Height'].mean()  ❷
Out[126]: 1.775
```

```
In [127]: s[0]  ❸
Out[127]: ('Smith', 45, 1.83, (0, 1))

In [128]: s[1]['Age']  ❹
Out[128]: 53
```

❶ Selecting a column by name.
❷ Calling a method on a selected column.
❸ Selecting a record.
❹ Selecting a field in a record.

In summary, structured arrays are a generalization of the regular `ndarray` object type in that the data type only has to be the same *per column*, like in tables in SQL databases. One advantage of structured arrays is that a single element of a column can be another multidimensional object and does not have to conform to the basic `NumPy` data types.

> **STRUCTURED ARRAYS**
>
> `NumPy` provides, in addition to regular arrays, structured (and record) arrays that allow the description and handling of table-like data structures with a variety of different data types per (named) column. They bring SQL table–like data structures to Python, with most of the benefits of regular `ndarray` objects (syntax, methods, performance).

Vectorization of Code

Vectorization is a strategy to get more compact code that is possibly executed faster. The fundamental idea is to conduct an operation on or to apply a function to a complex object "at once" and not by looping over the single elements of the object. In Python, functional programming tools such as `map()` and `filter()` provide some basic means for vectorization. However, `NumPy` has vectorization built in deep down in its core.

Basic Vectorization

As demonstrated in the previous section, simple mathematical operations—such as calculating the sum of all elements—can be implemented on `ndarray` objects

directly (via methods or universal functions). More general vectorized operations are also possible. For example, one can add two NumPy arrays element-wise as follows:

```
In [129]: np.random.seed(100)
          r = np.arange(12).reshape((4, 3))  ❶
          s = np.arange(12).reshape((4, 3)) * 0.5  ❷

In [130]: r  ❶
Out[130]: array([[ 0,  1,  2],
                 [ 3,  4,  5],
                 [ 6,  7,  8],
                 [ 9, 10, 11]])

In [131]: s  ❷
Out[131]: array([[0. , 0.5, 1. ],
                 [1.5, 2. , 2.5],
                 [3. , 3.5, 4. ],
                 [4.5, 5. , 5.5]])

In [132]: r + s  ❸
Out[132]: array([[ 0. ,  1.5,  3. ],
                 [ 4.5,  6. ,  7.5],
                 [ 9. , 10.5, 12. ],
                 [13.5, 15. , 16.5]])
```

❶ The first ndarray object with random numbers.

❷ The second ndarray object with random numbers.

❸ Element-wise addition as a vectorized operation (no looping).

NumPy also supports what is called *broadcasting*. This allows you to combine objects of different shape within a single operation. Previous examples have already made use of this. Consider the following examples:

```
In [133]: r + 3  ❶
Out[133]: array([[ 3,  4,  5],
                 [ 6,  7,  8],
                 [ 9, 10, 11],
                 [12, 13, 14]])

In [134]: 2 * r  ❷
Out[134]: array([[ 0,  2,  4],
                 [ 6,  8, 10],
                 [12, 14, 16],
                 [18, 20, 22]])
```

```
In [135]: 2 * r + 3  ❸
Out[135]: array([[ 3,  5,  7],
                 [ 9, 11, 13],
                 [15, 17, 19],
                 [21, 23, 25]])
```

❶ During scalar addition, the scalar is broadcast and added to every element.

❷ During scalar multiplication, the scalar is also broadcast to and multiplied with every element.

❸ This linear transformation combines both operations.

These operations work with differently shaped `ndarray` objects as well, up to a certain point:

```
In [136]: r
Out[136]: array([[ 0,  1,  2],
                 [ 3,  4,  5],
                 [ 6,  7,  8],
                 [ 9, 10, 11]])

In [137]: r.shape
Out[137]: (4, 3)

In [138]: s = np.arange(0, 12, 4)  ❶
          s  ❶
Out[138]: array([0, 4, 8])

In [139]: r + s  ❷
Out[139]: array([[ 0,  5, 10],
                 [ 3,  8, 13],
                 [ 6, 11, 16],
                 [ 9, 14, 19]])

In [140]: s = np.arange(0, 12, 3)  ❸
          s  ❸
Out[140]: array([0, 3, 6, 9])

In [141]: r + s  ❹

          ---------------------------------------
          ValueErrorTraceback (most recent call last)
          <ipython-input-141-1890b26ec965> in <module>()
          ----> 1 r + s  ❹

          ValueError: operands could not be broadcast together
                      with shapes (4,3) (4,)
```

```
In [142]: r.transpose() + s  ❺
Out[142]: array([[ 0,  6, 12, 18],
                 [ 1,  7, 13, 19],
                 [ 2,  8, 14, 20]])

In [143]: sr = s.reshape(-1, 1)  ❻
          sr
Out[143]: array([[0],
                 [3],
                 [6],
                 [9]])

In [144]: sr.shape  ❻
Out[144]: (4, 1)

In [145]: r + s.reshape(-1, 1)  ❻
Out[145]: array([[ 0,  1,  2],
                 [ 6,  7,  8],
                 [12, 13, 14],
                 [18, 19, 20]])
```

❶ A new one-dimensional `ndarray` object of length 3.

❷ The `r` (matrix) and `s` (vector) objects can be added straightforwardly.

❸ Another one-dimensional `ndarray` object of length 4.

❹ The length of the new `s` (vector) object is now different from the length of the second dimension of the `r` object.

❺ Transposing the `r` object again allows for the vectorized addition.

❻ Alternatively, the shape of `s` can be changed to `(4, 1)` to make the addition work (the results are different, however).

Often, custom-defined Python functions work with `ndarray` objects as well. If the implementation allows, arrays can be used with functions just as `int` or `float` objects can. Consider the following function:

```
In [146]: def f(x):
              return 3 * x + 5  ❶

In [147]: f(0.5)  ❷
Out[147]: 6.5

In [148]: f(r)  ❸
Out[148]: array([[ 5,  8, 11],
```

```
       [14, 17, 20],
       [23, 26, 29],
       [32, 35, 38]])
```

❶ A simple Python function implementing a linear transform on parameter `x`.

❷ The function `f()` applied to a Python `float` object.

❸ The same function applied to an `ndarray` object, resulting in a vectorized and element-wise evaluation of the function.

What `NumPy` does is to simply apply the function `f` to the object element-wise. In that sense, by using this kind of operation one does *not* avoid loops; one only avoids them on the Python level and delegates the looping to `NumPy`. On the `NumPy` level, looping over the `ndarray` object is taken care of by optimized code, most of it written in C and therefore generally faster than pure Python. This explains the "`secret`" behind the performance benefits of using `NumPy` for array-based use cases.

Memory Layout

When `ndarray` objects are initialized by using `np.zeros()`, as in "Multiple Dimensions", an optional argument for the memory layout is provided. This argument specifies, roughly speaking, which elements of an array get stored in memory next to each other (contiguously). When working with small arrays, this has hardly any measurable impact on the performance of array operations. However, when arrays get large, and depending on the (financial) algorithm to be implemented on them, the story might be different. This is when *memory layout* comes into play (see, for instance, Eli Bendersky's article "Memory Layout of Multi-Dimensional Arrays").

To illustrate the potential importance of the memory layout of arrays in science and finance, consider the following construction of multidimensional `ndarray` objects:

```
In [149]: x = np.random.standard_normal((1000000, 5))  ❶

In [150]: y = 2 * x + 3  ❷

In [151]: C = np.array((x, y), order='C')  ❸
```

```
In [152]: F = np.array((x, y), order='F')  ❹

In [153]: x = 0.0; y = 0.0  ❺

In [154]: C[:2].round(2)  ❻
Out[154]: array([[[-1.75,  0.34,  1.15, -0.25,  0.98],
                  [ 0.51,  0.22, -1.07, -0.19,  0.26],
                  [-0.46,  0.44, -0.58,  0.82,  0.67],
                  ...,
                  [-0.05,  0.14,  0.17,  0.33,  1.39],
                  [ 1.02,  0.3 , -1.23, -0.68, -0.87],
                  [ 0.83, -0.73,  1.03,  0.34, -0.46]],

                 [[-0.5 ,  3.69,  5.31,  2.5 ,  4.96],
                  [ 4.03,  3.44,  0.86,  2.62,  3.51],
                  [ 2.08,  3.87,  1.83,  4.63,  4.35],
                  ...,
                  [ 2.9 ,  3.28,  3.33,  3.67,  5.78],
                  [ 5.04,  3.6 ,  0.54,  1.65,  1.26],
                  [ 4.67,  1.54,  5.06,  3.69,  2.07]]])
```

❶ An `ndarray` object with large asymmetry in the two dimensions.

❷ A linear transform of the original object data.

❸ This creates a two-dimensional `ndarray` object with `C` order (row-major).

❹ This creates a two-dimensional `ndarray` object with `F` order (column-major).

❺ Memory is freed up (contingent on garbage collection).

❻ Some numbers from the `C` object.

Let's look at some fundamental examples and use cases for both types of `ndarray` objects and consider the speed with which they are executed given the different memory layouts:

```
In [155]: %timeit C.sum()  ❶
          4.36 ms ± 89.3 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)

In [156]: %timeit F.sum()  ❶
          4.21 ms ± 71.4 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)

In [157]: %timeit C.sum(axis=0)  ❷
          17.9 ms ± 776 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)

In [158]: %timeit C.sum(axis=1)  ❸
          35.1 ms ± 999 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)

In [159]: %timeit F.sum(axis=0)  ❷
```

```
            83.8 ms ± 2.63 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)

In [160]: %timeit F.sum(axis=1)  ❸
            67.9 ms ± 5.16 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)

In [161]: F = 0.0; C = 0.0
```

❶ Calculates the sum of all elements.
❷ Calculates the sums per row (“many”).
❸ Calculates the sums per columns (“few”).

We can summarize the performance results as follows:

- When calculating the sum of *all elements*, the memory layout does not really matter.
- The summing up over the `C`-ordered `ndarray` objects is faster both over rows and over columns (an *absolute* speed advantage).
- With the `C`-ordered (row-major) `ndarray` object, summing up over rows is *relatively* faster compared to summing up over columns.
- With the `F`-ordered (column-major) `ndarray` object, summing up over columns is *relatively* faster compared to summing up over rows.

Conclusion

`NumPy` is the package of choice for numerical computing in Python. The `ndarray` class is specifically designed to be convenient and efficient in the handling of (large) numerical data. Powerful methods and `NumPy` universal functions allow for vectorized code that mostly avoids slow loops on the Python level. Many approaches introduced in this chapter carry over to `pandas` and its `DataFrame` class as well (see Chapter 5).

Further Resources

Many helpful resources are provided at the `NumPy` website:

- *http://www.numpy.org/*

Good introductions to `NumPy` in book form are:

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

Chapter 5. Data Analysis with pandas

Data! Data! Data! I can't make bricks without clay!

—Sherlock Holmes

This chapter is about `pandas`, a library for data analysis with a focus on tabular data. `pandas` is a powerful tool that not only provides many useful classes and functions but also does a great job of wrapping functionality from other packages. The result is a user interface that makes data analysis, and in particular financial analysis, a convenient and efficient task.

This chapter covers the following fundamental data structures:

Object type	Meaning	Used for
`DataFrame`	2-dimensional data object with index	Tabular data organized in columns
`Series`	1-dimensional data object with index	Single (time) series of data

The chapter is organized as follows:

"The DataFrame Class"

This section starts by exploring the basic characteristics and capabilities of the `DataFrame` class of `pandas` by using simple and small data sets; it then shows how to transform a `NumPy` `ndarray` object into a `DataFrame` object.

"Basic Analytics" and "Basic Visualization"

Basic analytics and visualization capabilities are introduced in these sections (later chapters go deeper into these topics).

"The Series Class"

This rather brief section covers the `Series` class of `pandas`, which in a sense represents a special case of the `DataFrame` class with a single column of data

only.

"GroupBy Operations"

One of the strengths of the `DataFrame` class lies in grouping data according to a single or multiple columns. This section explores the grouping capabilities of `pandas`.

"Complex Selection"

This section illustrates how the use of (complex) conditions allows for the easy selection of data from a `DataFrame` object.

"Concatenation, Joining, and Merging"

The combining of different data sets into one is an important operation in data analysis. `pandas` provides different options to accomplish this task, as described in this section.

"Performance Aspects"

Like Python in general, `pandas` often provides multiple options to accomplish the same goal. This section takes a brief look at potential performance differences.

The DataFrame Class

At the core of `pandas` (and this chapter) is the `DataFrame`, a class designed to efficiently handle data in tabular form—i.e., data characterized by a columnar organization. To this end, the `DataFrame` class provides, for instance, column labeling as well as flexible indexing capabilities for the rows (records) of the data set, similar to a table in a relational database or an Excel spreadsheet.

This section covers some fundamental aspects of the `pandas` `DataFrame` class. The class is so complex and powerful that only a fraction of its capabilities can be presented here. Subsequent chapters provide more examples and shed light on different aspects.

First Steps with the DataFrame Class

On a fundamental level, the `DataFrame` class is designed to manage indexed and labeled data, not too different from a SQL database table or a worksheet in a spreadsheet application. Consider the following creation of a `DataFrame` object:

```
In [1]: import pandas as pd  ❶

In [2]: df = pd.DataFrame([10, 20, 30, 40],  ❷
                          columns=['numbers'],  ❸
                          index=['a', 'b', 'c', 'd'])  ❹

In [3]: df  ❺
Out[3]:    numbers
        a       10
        b       20
        c       30
        d       40
```

❶ Imports `pandas`.
❷ Defines the data as a `list` object.
❸ Specifies the column label.
❹ Specifies the index values/labels.
❺ Shows the data as well as column and index labels of the `DataFrame` object.

This simple example already shows some major features of the `DataFrame` class when it comes to storing data:

- Data itself can be provided in different shapes and types (`list`, `tuple`, `ndarray`, and `dict` objects are candidates).
- Data is organized in columns, which can have custom names (labels).
- There is an index that can take on different formats (e.g., numbers, strings, time information).

Working with a `DataFrame` object is in general pretty convenient and efficient with regard to the handling of the object, e.g., compared to regular `ndarray` objects, which are more specialized and more restricted when one wants to (say) enlarge an existing object. At the same time, `DataFrame` objects are often as computationally efficient as `ndarray` objects. The following are simple examples showing how typical operations on a `DataFrame` object work:

```
In [4]: df.index  ❶
Out[4]: Index(['a', 'b', 'c', 'd'], dtype='object')

In [5]: df.columns  ❷
Out[5]: Index(['numbers'], dtype='object')

In [6]: df.loc['c']  ❸
Out[6]: numbers    30
        Name: c, dtype: int64

In [7]: df.loc[['a', 'd']]  ❹
Out[7]:    numbers
        a       10
        d       40

In [8]: df.iloc[1:3]  ❺
Out[8]:    numbers
        b       20
        c       30

In [9]: df.sum()  ❻
Out[9]: numbers    100
        dtype: int64

In [10]: df.apply(lambda x: x ** 2)  ❼
Out[10]:    numbers
         a      100
         b      400
         c      900
         d     1600

In [11]: df ** 2  ❽
Out[11]:    numbers
         a      100
         b      400
         c      900
         d     1600
```

❶ The `index` attribute and `Index` object.

❷ The `columns` attribute and `Index` object.

❸ Selects the value corresponding to index `c`.

❹ Selects the two values corresponding to indices `a` and `d`.

❺ Selects the second and third rows via the index positions.

❻ Calculates the sum of the single column.

❼ Uses the `apply()` method to calculate squares in vectorized fashion.

❽ Applies vectorization directly as with `ndarray` objects.

Contrary to `NumPy ndarray` objects, enlarging the `DataFrame` object in both dimensions is possible:

```
In [12]: df['floats'] = (1.5, 2.5, 3.5, 4.5)  ❶

In [13]: df
Out[13]:    numbers  floats
         a       10     1.5
         b       20     2.5
         c       30     3.5
         d       40     4.5

In [14]: df['floats']  ❷
Out[14]: a    1.5
         b    2.5
         c    3.5
         d    4.5
         Name: floats, dtype: float64
```

❶ Adds a new column with `float` objects provided as a `tuple` object.

❷ Selects this column and shows its data and index labels.

A whole `DataFrame` object can also be taken to define a new column. In such a case, indices are aligned automatically:

```
In [15]: df['names'] = pd.DataFrame(['Yves', 'Sandra', 'Lilli', 'Henry'],
                                    index=['d', 'a', 'b', 'c'])  ❶

In [16]: df
Out[16]:    numbers  floats   names
         a       10     1.5  Sandra
         b       20     2.5   Lilli
         c       30     3.5   Henry
         d       40     4.5    Yves
```

❶ Another new column is created based on a `DataFrame` object.

Appending data works similarly. However, in the following example a side effect is seen that is usually to be avoided—namely, the index gets replaced by a simple range index:

```
In [17]: df.append({'numbers': 100, 'floats': 5.75, 'names': 'Jil'},
```

```
                      ignore_index=True)  ❶
Out[17]:    numbers  floats   names
         0       10    1.50  Sandra
         1       20    2.50   Lilli
         2       30    3.50   Henry
         3       40    4.50    Yves
         4      100    5.75     Jil

In [18]: df = df.append(pd.DataFrame({'numbers': 100, 'floats': 5.75,
                                      'names': 'Jil'}, index=['y',]))  ❷

In [19]: df
Out[19]:    numbers  floats   names
         a       10    1.50  Sandra
         b       20    2.50   Lilli
         c       30    3.50   Henry
         d       40    4.50    Yves
         y      100    5.75     Jil

In [20]: df = df.append(pd.DataFrame({'names': 'Liz'}, index=['z',]),
                        sort=False)  ❸

In [21]: df
Out[21]:    numbers  floats   names
         a     10.0    1.50  Sandra
         b     20.0    2.50   Lilli
         c     30.0    3.50   Henry
         d     40.0    4.50    Yves
         y    100.0    5.75     Jil
         z      NaN     NaN     Liz

In [22]: df.dtypes  ❹
Out[22]: numbers    float64
         floats     float64
         names       object
         dtype: object
```

❶ Appends a new row via a `dict` object; this is a temporary operation during which index information gets lost.

❷ Appends the row based on a `DataFrame` object with index information; the original index information is preserved.

❸ Appends an incomplete data row to the `DataFrame` object, resulting in `NaN` values.

❹ Returns the different `dtypes` of the single columns; this is similar to what's possible with structured `ndarray` objects.

Although there are now missing values, the majority of method calls will still work:

```
In [23]: df[['numbers', 'floats']].mean()  ❶
Out[23]: numbers    40.00
         floats      3.55
         dtype: float64

In [24]: df[['numbers', 'floats']].std()  ❷
Out[24]: numbers    35.355339
         floats      1.662077
         dtype: float64
```

❶ Calculates the mean over the two columns specified (ignoring rows with `NaN` values).

❷ Calculates the standard deviation over the two columns specified (ignoring rows with `NaN` values).

Second Steps with the DataFrame Class

The example in this subsection is based on an `ndarray` object with standard normally distributed random numbers. It explores further features such as a `DatetimeIndex` to manage time series data:

```
In [25]: import numpy as np

In [26]: np.random.seed(100)

In [27]: a = np.random.standard_normal((9, 4))

In [28]: a
Out[28]: array([[-1.74976547,  0.3426804 ,  1.1530358 , -0.25243604],
                [ 0.98132079,  0.51421884,  0.22117967, -1.07004333],
                [-0.18949583,  0.25500144, -0.45802699,  0.43516349],
                [-0.58359505,  0.81684707,  0.67272081, -0.10441114],
                [-0.53128038,  1.02973269, -0.43813562, -1.11831825],
                [ 1.61898166,  1.54160517, -0.25187914, -0.84243574],
                [ 0.18451869,  0.9370822 ,  0.73100034,  1.36155613],
                [-0.32623806,  0.05567601,  0.22239961, -1.443217  ],
                [-0.75635231,  0.81645401,  0.75044476, -0.45594693]])
```

Although one can construct `DataFrame` objects more directly (as seen before),

using an `ndarray` object is generally a good choice since `pandas` will retain the basic structure and will "only" add metainformation (e.g., index values). It also represents a typical use case for financial applications and scientific research in general. For example:

```
In [29]: df = pd.DataFrame(a)  ❶

In [30]: df
Out[30]:           0         1         2         3
         0 -1.749765  0.342680  1.153036 -0.252436
         1  0.981321  0.514219  0.221180 -1.070043
         2 -0.189496  0.255001 -0.458027  0.435163
         3 -0.583595  0.816847  0.672721 -0.104411
         4 -0.531280  1.029733 -0.438136 -1.118318
         5  1.618982  1.541605 -0.251879 -0.842436
         6  0.184519  0.937082  0.731000  1.361556
         7 -0.326238  0.055676  0.222400 -1.443217
         8 -0.756352  0.816454  0.750445 -0.455947
```

❶ Creates a `DataFrame` object from the `ndarray` object.

Table 5-1 lists the parameters that the `DataFrame()` function takes. In the table, "array-like" means a data structure similar to an `ndarray` object—a `list`, for example. `Index` is an instance of the `pandas Index` class.

Table 5-1. Parameters of DataFrame() function

Parameter	Format	Description
`data`	`ndarray/dict/DataFrame`	Data for `DataFrame`; `dict` can contain `Series`, `ndarray`, `list`
`index`	`Index`/array-like	Index to use; defaults to `range(`*n*`)`
`columns`	`Index`/array-like	Column headers to use; defaults to `range(`*n*`)`
`dtype`	`dtype`, default `None`	Data type to use/force; otherwise, it is inferred
`copy`	`bool`, default `None`	Copy data from inputs

As with structured arrays, and as seen before, `DataFrame` objects have column names that can be defined directly by assigning a `list` object with the right

number of elements. This illustrates that one can define/change the attributes of the `DataFrame` object easily:

```
In [31]: df.columns = ['No1', 'No2', 'No3', 'No4']  ❶

In [32]: df
Out[32]:         No1       No2       No3       No4
         0 -1.749765  0.342680  1.153036 -0.252436
         1  0.981321  0.514219  0.221180 -1.070043
         2 -0.189496  0.255001 -0.458027  0.435163
         3 -0.583595  0.816847  0.672721 -0.104411
         4 -0.531280  1.029733 -0.438136 -1.118318
         5  1.618982  1.541605 -0.251879 -0.842436
         6  0.184519  0.937082  0.731000  1.361556
         7 -0.326238  0.055676  0.222400 -1.443217
         8 -0.756352  0.816454  0.750445 -0.455947

In [33]: df['No2'].mean()  ❷
Out[33]: 0.7010330941456459
```

❶ Specifies the column labels via a `list` object.

❷ Picking a column is now made easy.

To work with financial time series data efficiently, one must be able to handle time indices well. This can also be considered a major strength of `pandas`. For example, assume that our nine data entries in the four columns correspond to month-end data, beginning in January 2019. A `DatetimeIndex` object is then generated with the `date_range()` function as follows:

```
In [34]: dates = pd.date_range('2019-1-1', periods=9, freq='M')  ❶

In [35]: dates
Out[35]: DatetimeIndex(['2019-01-31', '2019-02-28', '2019-03-31', '2019-04-30',
                        '2019-05-31', '2019-06-30', '2019-07-31', '2019-08-31',
                        '2019-09-30'],
                       dtype='datetime64[ns]', freq='M')
```

❶ Creates a `DatetimeIndex` object.

Table 5-2 lists the parameters that the `date_range()` function takes.

Table 5-2. Parameters of date_range() function

Parameter	Format	Description
`start`	`string/datetime`	Left bound for generating dates
`end`	`string/datetime`	Right bound for generating dates
`periods`	`integer/None`	Number of periods (if `start` or `end` is `None`)
`freq`	`string/DateOffset`	Frequency string, e.g., `5D` for 5 days
`tz`	`string/None`	Time zone name for localized index
`normalize`	`bool`, default `None`	Normalizes `start` and `end` to midnight
`name`	`string`, default `None`	Name of resulting index

The following code defines the just-created `DatetimeIndex` object as the relevant index object, making a time series of the original data set:

```
In [36]: df.index = dates

In [37]: df
Out[37]:                  No1       No2       No3       No4
         2019-01-31 -1.749765  0.342680  1.153036 -0.252436
         2019-02-28  0.981321  0.514219  0.221180 -1.070043
         2019-03-31 -0.189496  0.255001 -0.458027  0.435163
         2019-04-30 -0.583595  0.816847  0.672721 -0.104411
         2019-05-31 -0.531280  1.029733 -0.438136 -1.118318
         2019-06-30  1.618982  1.541605 -0.251879 -0.842436
         2019-07-31  0.184519  0.937082  0.731000  1.361556
         2019-08-31 -0.326238  0.055676  0.222400 -1.443217
         2019-09-30 -0.756352  0.816454  0.750445 -0.455947
```

When it comes to the generation of `DatetimeIndex` objects with the help of the `date_range()` function, there are a number of choices for the frequency parameter `freq`. Table 5-3 lists all the options.

Table 5-3. Frequency parameter values for date_range() function

Alias	Description
`B`	Business day frequency

`C`	Custom business day frequency (experimental)
`D`	Calendar day frequency
`W`	Weekly frequency
`M`	Month end frequency
`BM`	Business month end frequency
`MS`	Month start frequency
`BMS`	Business month start frequency
`Q`	Quarter end frequency
`BQ`	Business quarter end frequency
`QS`	Quarter start frequency
`BQS`	Business quarter start frequency
`A`	Year end frequency
`BA`	Business year end frequency
`AS`	Year start frequency
`BAS`	Business year start frequency
`H`	Hourly frequency
`T`	Minutely frequency
`S`	Secondly frequency
`L`	Milliseconds
`U`	Microseconds

In some circumstances, it pays off to have access to the original data set in the form of the `ndarray` object. The `values` attribute provides direct access to it:

```
In [38]: df.values
Out[38]: array([[-1.74976547,  0.3426804 ,  1.1530358 , -0.25243604],
                [ 0.98132079,  0.51421884,  0.22117967, -1.07004333],
```

```
                [-0.18949583,  0.25500144, -0.45802699,  0.43516349],
                [-0.58359505,  0.81684707,  0.67272081, -0.10441114],
                [-0.53128038,  1.02973269, -0.43813562, -1.11831825],
                [ 1.61898166,  1.54160517, -0.25187914, -0.84243574],
                [ 0.18451869,  0.9370822 ,  0.73100034,  1.36155613],
                [-0.32623806,  0.05567601,  0.22239961, -1.443217  ],
                [-0.75635231,  0.81645401,  0.75044476, -0.45594693]])

In [39]: np.array(df)
Out[39]: array([[-1.74976547,  0.3426804 ,  1.1530358 , -0.25243604],
                [ 0.98132079,  0.51421884,  0.22117967, -1.07004333],
                [-0.18949583,  0.25500144, -0.45802699,  0.43516349],
                [-0.58359505,  0.81684707,  0.67272081, -0.10441114],
                [-0.53128038,  1.02973269, -0.43813562, -1.11831825],
                [ 1.61898166,  1.54160517, -0.25187914, -0.84243574],
                [ 0.18451869,  0.9370822 ,  0.73100034,  1.36155613],
                [-0.32623806,  0.05567601,  0.22239961, -1.443217  ],
                [-0.75635231,  0.81645401,  0.75044476, -0.45594693]])
```

ARRAYS AND DATAFRAMES

One can generate a `DataFrame` object from an `ndarray` object, but one can also easily generate an `ndarray` object out of a `DataFrame` by using the `values` attribute of the `DataFrame` class or the function `np.array()` of `NumPy`.

Basic Analytics

Like the `NumPy` `ndarray` class, the `pandas` `DataFrame` class has a multitude of convenience methods built in. As a starter, consider the methods `info()` and `describe()`:

```
In [40]: df.info()  ❶
         <class 'pandas.core.frame.DataFrame'>
         DatetimeIndex: 9 entries, 2019-01-31 to 2019-09-30
         Freq: M
         Data columns (total 4 columns):
         No1    9 non-null float64
         No2    9 non-null float64
         No3    9 non-null float64
         No4    9 non-null float64
         dtypes: float64(4)
         memory usage: 360.0 bytes
```

```
In [41]: df.describe()  ❷
Out[41]:              No1       No2       No3       No4
         count  9.000000  9.000000  9.000000  9.000000
         mean  -0.150212  0.701033  0.289193 -0.387788
         std    0.988306  0.457685  0.579920  0.877532
         min   -1.749765  0.055676 -0.458027 -1.443217
         25%   -0.583595  0.342680 -0.251879 -1.070043
         50%   -0.326238  0.816454  0.222400 -0.455947
         75%    0.184519  0.937082  0.731000 -0.104411
         max    1.618982  1.541605  1.153036  1.361556
```

❶ Provides metainformation regarding the data, columns, and index.

❷ Provides helpful summary statistics per column (for numerical data).

In addition, one can easily get the column-wise or row-wise sums, means, and cumulative sums:

```
In [43]: df.sum()  ❶
Out[43]: No1   -1.351906
         No2    6.309298
         No3    2.602739
         No4   -3.490089
         dtype: float64

In [44]: df.mean()  ❷
Out[44]: No1   -0.150212
         No2    0.701033
         No3    0.289193
         No4   -0.387788
         dtype: float64

In [45]: df.mean(axis=0)  ❷
Out[45]: No1   -0.150212
         No2    0.701033
         No3    0.289193
         No4   -0.387788
         dtype: float64

In [46]: df.mean(axis=1)  ❸
Out[46]: 2019-01-31   -0.126621
         2019-02-28    0.161669
         2019-03-31    0.010661
         2019-04-30    0.200390
         2019-05-31   -0.264500
         2019-06-30    0.516568
         2019-07-31    0.803539
         2019-08-31   -0.372845
         2019-09-30    0.088650
```

```
          Freq: M, dtype: float64

In [47]: df.cumsum()  ❹
Out[47]:                  No1       No2       No3       No4
          2019-01-31 -1.749765  0.342680  1.153036 -0.252436
          2019-02-28 -0.768445  0.856899  1.374215 -1.322479
          2019-03-31 -0.957941  1.111901  0.916188 -0.887316
          2019-04-30 -1.541536  1.928748  1.588909 -0.991727
          2019-05-31 -2.072816  2.958480  1.150774 -2.110045
          2019-06-30 -0.453834  4.500086  0.898895 -2.952481
          2019-07-31 -0.269316  5.437168  1.629895 -1.590925
          2019-08-31 -0.595554  5.492844  1.852294 -3.034142
          2019-09-30 -1.351906  6.309298  2.602739 -3.490089
```

❶ Column-wise sum.

❷ Column-wise mean.

❸ Row-wise mean.

❹ Column-wise cumulative sum (starting at first index position).

`DataFrame` objects also understand `NumPy` universal functions, as expected:

```
In [48]: np.mean(df)  ❶
Out[48]: No1   -0.150212
          No2    0.701033
          No3    0.289193
          No4   -0.387788
          dtype: float64

In [49]: np.log(df)  ❷
Out[49]:                  No1       No2       No3       No4
          2019-01-31       NaN -1.070957  0.142398       NaN
          2019-02-28 -0.018856 -0.665106 -1.508780       NaN
          2019-03-31       NaN -1.366486       NaN -0.832033
          2019-04-30       NaN -0.202303 -0.396425       NaN
          2019-05-31       NaN  0.029299       NaN       NaN
          2019-06-30  0.481797  0.432824       NaN       NaN
          2019-07-31 -1.690005 -0.064984 -0.313341  0.308628
          2019-08-31       NaN -2.888206 -1.503279       NaN
          2019-09-30       NaN -0.202785 -0.287089       NaN

In [50]: np.sqrt(abs(df))  ❸
Out[50]:                  No1       No2       No3       No4
          2019-01-31  1.322787  0.585389  1.073795  0.502430
          2019-02-28  0.990616  0.717091  0.470297  1.034429
          2019-03-31  0.435311  0.504977  0.676777  0.659669
          2019-04-30  0.763934  0.903796  0.820196  0.323127
          2019-05-31  0.728890  1.014757  0.661918  1.057506
```

```
2019-06-30  1.272392  1.241614  0.501876  0.917843
2019-07-31  0.429556  0.968030  0.854986  1.166857
2019-08-31  0.571173  0.235958  0.471593  1.201340
2019-09-30  0.869685  0.903578  0.866282  0.675238

In [51]: np.sqrt(abs(df)).sum()  ❹
Out[51]: No1    7.384345
         No2    7.075190
         No3    6.397719
         No4    7.538440
         dtype: float64

In [52]: 100 * df + 100  ❺
Out[52]:                    No1         No2         No3         No4
         2019-01-31  -74.976547  134.268040  215.303580   74.756396
         2019-02-28  198.132079  151.421884  122.117967   -7.004333
         2019-03-31   81.050417  125.500144   54.197301  143.516349
         2019-04-30   41.640495  181.684707  167.272081   89.558886
         2019-05-31   46.871962  202.973269   56.186438  -11.831825
         2019-06-30  261.898166  254.160517   74.812086   15.756426
         2019-07-31  118.451869  193.708220  173.100034  236.155613
         2019-08-31   67.376194  105.567601  122.239961  -44.321700
         2019-09-30   24.364769  181.645401  175.044476   54.405307
```

❶ Column-wise mean.

❷ Element-wise natural logarithm; a warning is raised but the calculation runs through, resulting in multiple `NaN` values.

❸ Element-wise square root for the absolute values …

❹ … and column-wise mean values for the results.

❺ A linear transform of the numerical data.

> **NUMPY UNIVERSAL FUNCTIONS**
>
> In general, one can apply `NumPy` universal functions to `pandas` `DataFrame` objects whenever they could be applied to an `ndarray` object containing the same type of data.

`pandas` is quite error tolerant, in the sense that it captures errors and just puts a `NaN` value where the respective mathematical operation fails. Not only this, but as briefly shown before, one can also work with such incomplete data sets as if they were complete in a number of cases. This comes in handy, since reality is characterized by incomplete data sets more often than one might wish.

Basic Visualization

Plotting of data is only one line of code away in general, once the data is stored in a `DataFrame` object (see Figure 5-1):

```
In [53]: from pylab import plt, mpl  ❶
         plt.style.use('seaborn')  ❶
         mpl.rcParams['font.family'] = 'serif'  ❶
         %matplotlib inline

In [54]: df.cumsum().plot(lw=2.0, figsize=(10, 6));  ❷
```

❶ Customizing the plotting style.

❷ Plotting the cumulative sums of the four columns as a line plot.

Basically, `pandas` provides a wrapper around `matplotlib` (see Chapter 7), specifically designed for `DataFrame` objects. Table 5-4 lists the parameters that the `plot()` method takes.

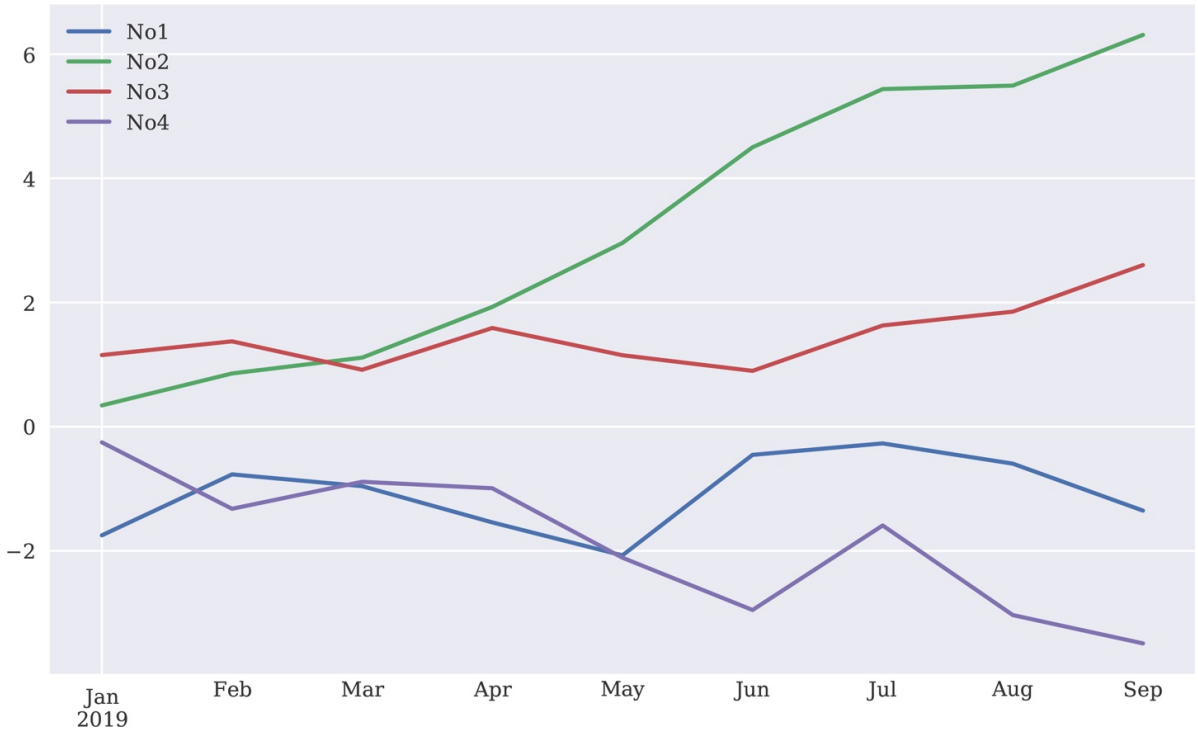

Figure 5-1. Line plot of a DataFrame object

Table 5-4. Parameters of plot() method

Parameter	Format	Description

`x`	label/position, default `None`	Only used when column values are x-ticks
`y`	label/position, default `None`	Only used when column values are y-ticks
`subplots`	boolean, default `False`	Plot columns in subplots
`sharex`	boolean, default `True`	Share the x-axis
`sharey`	boolean, default `False`	Share the y-axis
`use_index`	boolean, default `True`	Use `DataFrame.index` as x-ticks
`stacked`	boolean, default `False`	Stack (only for bar plots)
`sort_columns`	boolean, default `False`	Sort columns alphabetically before plotting
`title`	string, default `None`	Title for the plot
`grid`	boolean, default `False`	Show horizontal and vertical grid lines
`legend`	boolean, default `True`	Show legend of labels
`ax`	`matplotlib` axis object	`matplotlib` axis object to use for plotting
`style`	string or list/dictionary	Line plotting style (for each column)
`kind`	string (e.g., `"line"`, `"bar"`, `"barh"`, `"kde"`, `"density"`)	Type of plot
`logx`	boolean, default `False`	Use logarithmic scaling of x-axis
`logy`	boolean, default `False`	Use logarithmic scaling of y-axis
`xticks`	sequence, default `Index`	X-ticks for the plot
`yticks`	sequence, default `Values`	Y-ticks for the plot
`xlim`	2-tuple, list	Boundaries for x-axis
`ylim`	2-tuple, list	Boundaries for y-axis

`rot`	integer, default `None`	Rotation of x-ticks
`secondary_y`	boolean/sequence, default `False`	Plot on secondary y-axis
`mark_right`	boolean, default `True`	Automatic labeling of secondary axis
`colormap`	string/colormap object, default `None`	Color map to use for plotting
`kwds`	keywords	Options to pass to `matplotlib`

As another example, consider a bar plot of the same data (see Figure 5-2):

```
In [55]: df.plot.bar(figsize=(10, 6), rot=15);  ❶
         # df.plot(kind='bar', figsize=(10, 6))  ❷
```

❶ Plots the bar chart via `.plot.bar()`.

❷ Alternative syntax: uses the `kind` parameter to change the plot type.

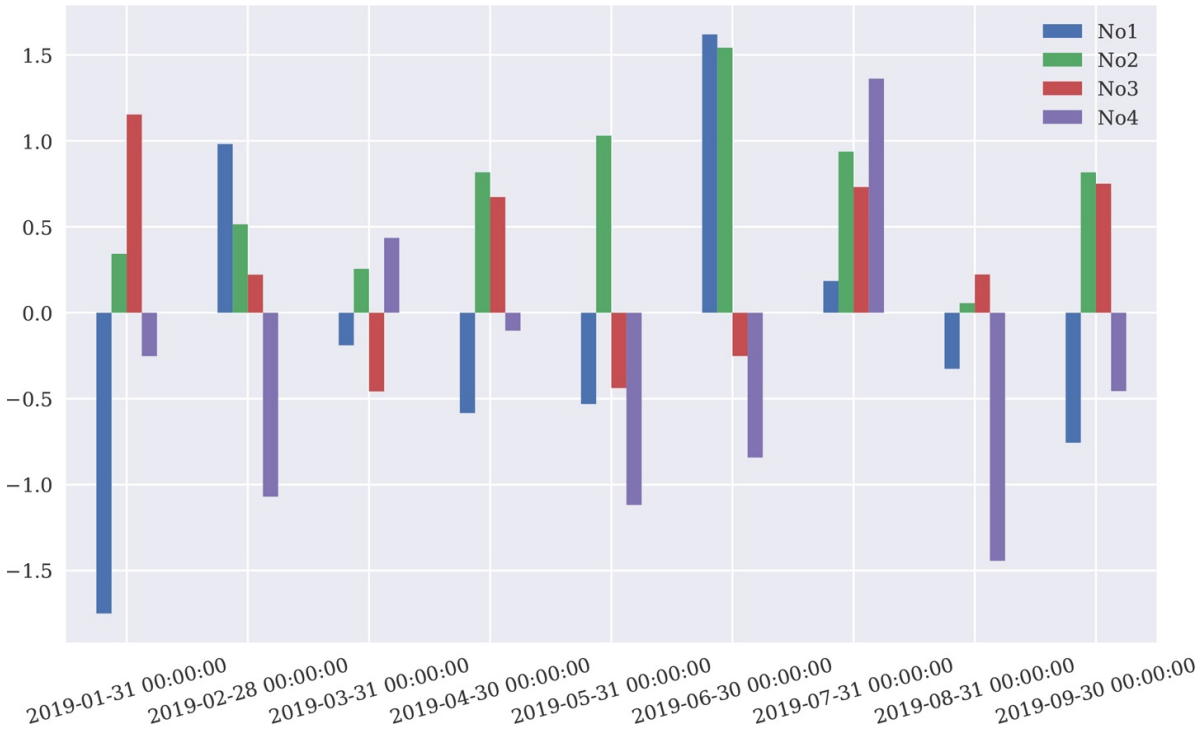

Figure 5-2. Bar plot of a DataFrame object

The Series Class

So far this chapter has worked mainly with the `pandas` `DataFrame` class. `Series` is another important class that comes with `pandas`. It is characterized by the fact that it has only a single column of data. In that sense, it is a specialization of the `DataFrame` class that shares many but not all of its characteristics and capabilities. A `Series` object is obtained when a single column is selected from a multicolumn `DataFrame` object:

```
In [56]: type(df)
Out[56]: pandas.core.frame.DataFrame

In [57]: S = pd.Series(np.linspace(0, 15, 7), name='series')

In [58]: S
Out[58]: 0     0.0
         1     2.5
         2     5.0
         3     7.5
         4    10.0
         5    12.5
         6    15.0
         Name: series, dtype: float64

In [59]: type(S)
Out[59]: pandas.core.series.Series

In [60]: s = df['No1']

In [61]: s
Out[61]: 2019-01-31   -1.749765
         2019-02-28    0.981321
         2019-03-31   -0.189496
         2019-04-30   -0.583595
         2019-05-31   -0.531280
         2019-06-30    1.618982
         2019-07-31    0.184519
         2019-08-31   -0.326238
         2019-09-30   -0.756352
         Freq: M, Name: No1, dtype: float64

In [62]: type(s)
Out[62]: pandas.core.series.Series
```

The main `DataFrame` methods are available for `Series` objects as well. For illustration, consider the `mean()` and `plot()` methods (see Figure 5-3):

```
In [63]: s.mean()
Out[63]: -0.15021177307319458

In [64]: s.plot(lw=2.0, figsize=(10, 6));
```

Figure 5-3. Line plot of a Series object

GroupBy Operations

pandas has powerful and flexible grouping capabilities. They work similarly to grouping in SQL as well as pivot tables in Microsoft Excel. To have something to group by one can add, for instance, a column indicating the quarter the respective data of the index belongs to:

```
In [65]: df['Quarter'] = ['Q1', 'Q1', 'Q1', 'Q2', 'Q2',
                          'Q2', 'Q3', 'Q3', 'Q3']
         df
Out[65]:                 No1       No2       No3       No4 Quarter
         2019-01-31 -1.749765  0.342680  1.153036 -0.252436      Q1
         2019-02-28  0.981321  0.514219  0.221180 -1.070043      Q1
         2019-03-31 -0.189496  0.255001 -0.458027  0.435163      Q1
         2019-04-30 -0.583595  0.816847  0.672721 -0.104411      Q2
         2019-05-31 -0.531280  1.029733 -0.438136 -1.118318      Q2
         2019-06-30  1.618982  1.541605 -0.251879 -0.842436      Q2
```

```
2019-07-31  0.184519  0.937082  0.731000  1.361556      Q3
2019-08-31 -0.326238  0.055676  0.222400 -1.443217      Q3
2019-09-30 -0.756352  0.816454  0.750445 -0.455947      Q3
```

The following code groups by the `Quarter` column and outputs statistics for the single groups:

```
In [66]: groups = df.groupby('Quarter')  ❶

In [67]: groups.size()  ❷
Out[67]: Quarter
         Q1    3
         Q2    3
         Q3    3
         dtype: int64

In [68]: groups.mean()  ❸
Out[68]:               No1       No2       No3       No4
         Quarter
         Q1      -0.319314  0.370634  0.305396 -0.295772
         Q2       0.168035  1.129395 -0.005765 -0.688388
         Q3      -0.299357  0.603071  0.567948 -0.179203

In [69]: groups.max()  ❹
Out[69]:               No1       No2       No3       No4
         Quarter
         Q1       0.981321  0.514219  1.153036  0.435163
         Q2       1.618982  1.541605  0.672721 -0.104411
         Q3       0.184519  0.937082  0.750445  1.361556

In [70]: groups.aggregate([min, max]).round(2)  ❺
Out[70]:           No1         No2         No3         No4
                   min   max   min   max   min   max   min   max
         Quarter
         Q1      -1.75  0.98  0.26  0.51 -0.46  1.15 -1.07  0.44
         Q2      -0.58  1.62  0.82  1.54 -0.44  0.67 -1.12 -0.10
         Q3      -0.76  0.18  0.06  0.94  0.22  0.75 -1.44  1.36
```

❶ Groups according to the `Quarter` column.

❷ Gives the number of rows in each group.

❸ Gives the mean per column.

❹ Gives the maximum value per column.

❺ Gives both the minimum and maximum values per column.

Grouping can also be done with multiple columns. To this end, another column,

indicating whether the month of the index date is odd or even, is introduced:

```
In [71]: df['Odd_Even'] = ['Odd', 'Even', 'Odd', 'Even', 'Odd', 'Even',
                           'Odd', 'Even', 'Odd']

In [72]: groups = df.groupby(['Quarter', 'Odd_Even'])

In [73]: groups.size()
Out[73]: Quarter  Odd_Even
         Q1       Even       1
                  Odd        2
         Q2       Even       2
                  Odd        1
         Q3       Even       1
                  Odd        2
         dtype: int64

In [74]: groups[['No1', 'No4']].aggregate([sum, np.mean])
Out[74]:                          No1                 No4
                                  sum      mean       sum      mean
         Quarter Odd_Even
         Q1      Even       0.981321  0.981321 -1.070043 -1.070043
                 Odd       -1.939261 -0.969631  0.182727  0.091364
         Q2      Even       1.035387  0.517693 -0.946847 -0.473423
                 Odd       -0.531280 -0.531280 -1.118318 -1.118318
         Q3      Even      -0.326238 -0.326238 -1.443217 -1.443217
                 Odd       -0.571834 -0.285917  0.905609  0.452805
```

This concludes the introduction to `pandas` and the use of `DataFrame` objects. Subsequent chapters apply this tool set to real financial data sets.

Complex Selection

Often, data selection is accomplished by formulation of conditions on column values, and potentially combining multiple such conditions logically. Consider the following data set:

```
In [75]: data = np.random.standard_normal((10, 2))  ❶

In [76]: df = pd.DataFrame(data, columns=['x', 'y'])  ❷

In [77]: df.info()  ❸
         <class 'pandas.core.frame.DataFrame'>
         RangeIndex: 10 entries, 0 to 9
```

```
Data columns (total 2 columns):
x    10 non-null float64
y    10 non-null float64
dtypes: float64(2)
memory usage: 240.0 bytes
```

```
In [78]: df.head()  ❸
Out[78]:          x         y
         0  1.189622 -1.690617
         1 -1.356399 -1.232435
         2 -0.544439 -0.668172
         3  0.007315 -0.612939
         4  1.299748 -1.733096
```

```
In [79]: df.tail()  ❹
Out[79]:          x         y
         5 -0.983310  0.357508
         6 -1.613579  1.470714
         7 -1.188018 -0.549746
         8 -0.940046 -0.827932
         9  0.108863  0.507810
```

❶ `ndarray` object with standard normally distributed random numbers.

❷ `DataFrame` object with the same random numbers.

❸ The first five rows via the `head()` method.

❹ The final five rows via the `tail()` method.

The following code illustrates the application of Python's comparison operators and logical operators on values in the two columns:

```
In [80]: df['x'] > 0.5  ❶
Out[80]: 0     True
         1    False
         2    False
         3    False
         4     True
         5    False
         6    False
         7    False
         8    False
         9    False
         Name: x, dtype: bool
```

```
In [81]: (df['x'] > 0) & (df['y'] < 0)  ❷
Out[81]: 0     True
         1    False
```

```
2    False
3     True
4     True
5    False
6    False
7    False
8    False
9    False
dtype: bool

In [82]: (df['x'] > 0) | (df['y'] < 0)  ❸
Out[82]: 0     True
         1     True
         2     True
         3     True
         4     True
         5    False
         6    False
         7     True
         8     True
         9     True
         dtype: bool
```

❶ Check whether value in column `x` is greater than 0.5.

❷ Check whether value in column `x` is positive *and* value in column `y` is negative.

❸ Check whether value in column `x` is positive *or* value in column `y` is negative.

Using the resulting Boolean `Series` objects, complex data (row) selection is straightforward. Alternatively, one can use the `query()` method and pass the conditions as `str` objects:

```
In [83]: df[df['x'] > 0]  ❶
Out[83]:           x         y
         0  1.189622 -1.690617
         3  0.007315 -0.612939
         4  1.299748 -1.733096
         9  0.108863  0.507810

In [84]: df.query('x > 0')  ❶
Out[84]:           x         y
         0  1.189622 -1.690617
         3  0.007315 -0.612939
         4  1.299748 -1.733096
```

```
          9  0.108863  0.507810

In [85]: df[(df['x'] > 0) & (df['y'] < 0)]  ❷
Out[85]:           x         y
          0  1.189622 -1.690617
          3  0.007315 -0.612939
          4  1.299748 -1.733096

In [86]: df.query('x > 0 & y < 0')  ❷
Out[86]:           x         y
          0  1.189622 -1.690617
          3  0.007315 -0.612939
          4  1.299748 -1.733096

In [87]: df[(df.x > 0) | (df.y < 0)]  ❸
Out[87]:           x         y
          0  1.189622 -1.690617
          1 -1.356399 -1.232435
          2 -0.544439 -0.668172
          3  0.007315 -0.612939
          4  1.299748 -1.733096
          7 -1.188018 -0.549746
          8 -0.940046 -0.827932
          9  0.108863  0.507810
```

❶ All rows for which the value in column `x` is greater than 0.5.

❷ All rows for which the value in column `x` is positive *and* the value in column `y` is negative.

❸ All rows for which the value in column `x` is positive *or* the value in column `y` is negative (columns are accessed here via the respective attributes).

Comparison operators can also be applied to complete `DataFrame` objects at once:

```
In [88]: df > 0  ❶
Out[88]:        x      y
          0   True  False
          1  False  False
          2  False  False
          3   True  False
          4   True  False
          5  False   True
          6  False   True
          7  False  False
          8  False  False
          9   True   True
```

```
In [89]: df[df > 0]  ❷
Out[89]:           x         y
         0  1.189622       NaN
         1       NaN       NaN
         2       NaN       NaN
         3  0.007315       NaN
         4  1.299748       NaN
         5       NaN  0.357508
         6       NaN  1.470714
         7       NaN       NaN
         8       NaN       NaN
         9  0.108863  0.507810
```

❶ Which values in the `DataFrame` object are positive?

❷ Select all such values and put a `NaN` in all other places.

Concatenation, Joining, and Merging

This section walks through different approaches to combine two simple data sets in the form of `DataFrame` objects. The two simple data sets are:

```
In [90]: df1 = pd.DataFrame(['100', '200', '300', '400'],
                            index=['a', 'b', 'c', 'd'],
                            columns=['A',])

In [91]: df1
Out[91]:      A
         a  100
         b  200
         c  300
         d  400

In [92]: df2 = pd.DataFrame(['200', '150', '50'],
                            index=['f', 'b', 'd'],
                            columns=['B',])

In [93]: df2
Out[93]:      B
         f  200
         b  150
         d   50
```

Concatenation

Concatenation or *appending* basically means that rows are added from one `DataFrame` object to another one. This can be accomplished via the `append()` method or via the `pd.concat()` function. A major consideration is how the index values are handled:

```
In [94]: df1.append(df2, sort=False)  ❶
Out[94]:      A    B
          a  100  NaN
          b  200  NaN
          c  300  NaN
          d  400  NaN
          f  NaN  200
          b  NaN  150
          d  NaN   50

In [95]: df1.append(df2, ignore_index=True, sort=False)  ❷
Out[95]:      A    B
          0  100  NaN
          1  200  NaN
          2  300  NaN
          3  400  NaN
          4  NaN  200
          5  NaN  150
          6  NaN   50

In [96]: pd.concat((df1, df2), sort=False)  ❸
Out[96]:      A    B
          a  100  NaN
          b  200  NaN
          c  300  NaN
          d  400  NaN
          f  NaN  200
          b  NaN  150
          d  NaN   50

In [97]: pd.concat((df1, df2), ignore_index=True, sort=False)  ❹
Out[97]:      A    B
          0  100  NaN
          1  200  NaN
          2  300  NaN
          3  400  NaN
          4  NaN  200
          5  NaN  150
          6  NaN   50
```

❶ Appends data from `df2` to `df1` as new rows.

❷ Does the same but ignores the indices.
❸ Has the same effect as the first append operation.
❹ Has the same effect as the second append operation.

Joining

When joining the two data sets, the sequence of the `DataFrame` objects also matters but in a different way. Only the index values from the first `DataFrame` object are used. This default behavior is called a *left join*:

```
In [98]: df1.join(df2)  ❶
Out[98]:      A    B
          a  100  NaN
          b  200  150
          c  300  NaN
          d  400   50

In [99]: df2.join(df1)  ❷
Out[99]:      B    A
          f  200  NaN
          b  150  200
          d   50  400
```

❶ Index values of `df1` are relevant.
❷ Index values of `df2` are relevant.

There are a total of four different join methods available, each leading to a different behavior with regard to how index values and the corresponding data rows are handled:

```
In [100]: df1.join(df2, how='left')  ❶
Out[100]:      A    B
           a  100  NaN
           b  200  150
           c  300  NaN
           d  400   50

In [101]: df1.join(df2, how='right')  ❷
Out[101]:      A    B
           f  NaN  200
           b  200  150
           d  400   50
```

```
In [102]: df1.join(df2, how='inner')  ❸
Out[102]:      A    B
          b  200  150
          d  400   50

In [103]: df1.join(df2, how='outer')  ❹
Out[103]:      A    B
          a  100  NaN
          b  200  150
          c  300  NaN
          d  400   50
          f  NaN  200
```

❶ Left join is the default operation.

❷ Right join is the same as reversing the sequence of the `DataFrame` objects.

❸ Inner join only preserves those index values found in both indices.

❹ Outer join preserves all index values from both indices.

A join can also happen based on an empty `DataFrame` object. In this case, the columns are created *sequentially*, leading to behavior similar to a left join:

```
In [104]: df = pd.DataFrame()

In [105]: df['A'] = df1['A']  ❶

In [106]: df
Out[106]:      A
          a  100
          b  200
          c  300
          d  400

In [107]: df['B'] = df2  ❷

In [108]: df
Out[108]:      A    B
          a  100  NaN
          b  200  150
          c  300  NaN
          d  400   50
```

❶ `df1` as first column `A`.

❷ `df2` as second column `B`.

Making use of a dictionary to combine the data sets yields a result similar to an

outer join since the columns are created *simultaneously*:

```
In [109]: df = pd.DataFrame({'A': df1['A'], 'B': df2['B']})  ❶

In [110]: df
Out[110]:      A    B
           a  100  NaN
           b  200  150
           c  300  NaN
           d  400   50
           f  NaN  200
```

❶ The columns of the `DataFrame` objects are used as values in the `dict` object.

Merging

While a join operation takes place based on the indices of the `DataFrame` objects to be joined, a merge operation typically takes place on a column shared between the two data sets. To this end, a new column `C` is added to both original `DataFrame` objects:

```
In [111]: c = pd.Series([250, 150, 50], index=['b', 'd', 'c'])
          df1['C'] = c
          df2['C'] = c

In [112]: df1
Out[112]:      A      C
           a  100    NaN
           b  200  250.0
           c  300   50.0
           d  400  150.0

In [113]: df2
Out[113]:      B      C
           f  200    NaN
           b  150  250.0
           d   50  150.0
```

By default, the merge operation in this case takes place based on the single shared column `C`. Other options are available, however, such as an *outer* merge:

```
In [114]: pd.merge(df1, df2)  ❶
Out[114]:      A      C    B
```

```
          0  100    NaN  200
          1  200  250.0  150
          2  400  150.0   50

In [115]: pd.merge(df1, df2, on='C')  ❶
Out[115]:      A      C    B
          0  100    NaN  200
          1  200  250.0  150
          2  400  150.0   50

In [116]: pd.merge(df1, df2, how='outer')  ❷
Out[116]:      A      C    B
          0  100    NaN  200
          1  200  250.0  150
          2  300   50.0  NaN
          3  400  150.0   50
```

❶ The default merge on column `C`.

❷ An outer merge is also possible, preserving all data rows.

Many more types of merge operations are available, a few of which are illustrated in the following code:

```
In [117]: pd.merge(df1, df2, left_on='A', right_on='B')
Out[117]:      A    C_x    B  C_y
          0  200  250.0  200  NaN

In [118]: pd.merge(df1, df2, left_on='A', right_on='B', how='outer')
Out[118]:      A    C_x    B    C_y
          0  100    NaN  NaN    NaN
          1  200  250.0  200    NaN
          2  300   50.0  NaN    NaN
          3  400  150.0  NaN    NaN
          4  NaN    NaN  150  250.0
          5  NaN    NaN   50  150.0

In [119]: pd.merge(df1, df2, left_index=True, right_index=True)
Out[119]:      A    C_x    B    C_y
          b  200  250.0  150  250.0
          d  400  150.0   50  150.0

In [120]: pd.merge(df1, df2, on='C', left_index=True)
Out[120]:      A      C    B
          f  100    NaN  200
          b  200  250.0  150
          d  400  150.0   50

In [121]: pd.merge(df1, df2, on='C', right_index=True)
```

```
Out[121]:      A      C    B
          a  100    NaN  200
          b  200  250.0  150
          d  400  150.0   50

In [122]: pd.merge(df1, df2, on='C', left_index=True, right_index=True)
Out[122]:      A      C    B
          b  200  250.0  150
          d  400  150.0   50
```

Performance Aspects

Many examples in this chapter illustrate that there are often multiple options to achieve the same goal with `pandas`. This section compares such options for adding up two columns element-wise. First, the data set, generated with `NumPy`:

```
In [123]: data = np.random.standard_normal((1000000, 2))  ❶

In [124]: data.nbytes  ❶
Out[124]: 16000000

In [125]: df = pd.DataFrame(data, columns=['x', 'y'])  ❷

In [126]: df.info()  ❷
          <class 'pandas.core.frame.DataFrame'>
          RangeIndex: 1000000 entries, 0 to 999999
          Data columns (total 2 columns):
          x    1000000 non-null float64
          y    1000000 non-null float64
          dtypes: float64(2)
          memory usage: 15.3 MB
```

❶ The `ndarray` object with random numbers.

❷ The `DataFrame` object with the random numbers.

Second, some options to accomplish the task at hand with performance values:

```
In [127]: %time res = df['x'] + df['y']  ❶
          CPU times: user 7.35 ms, sys: 7.43 ms, total: 14.8 ms
          Wall time: 7.48 ms

In [128]: res[:3]
Out[128]: 0    0.387242
          1   -0.969343
```

```
2   -0.863159
dtype: float64

In [129]: %time res = df.sum(axis=1)  ❷
CPU times: user 130 ms, sys: 30.6 ms, total: 161 ms
Wall time: 101 ms

In [130]: res[:3]
Out[130]: 0    0.387242
1   -0.969343
2   -0.863159
dtype: float64

In [131]: %time res = df.values.sum(axis=1)  ❸
CPU times: user 50.3 ms, sys: 2.75 ms, total: 53.1 ms
Wall time: 27.9 ms

In [132]: res[:3]
Out[132]: array([ 0.3872424 , -0.96934273, -0.86315944])

In [133]: %time res = np.sum(df, axis=1)  ❹
CPU times: user 127 ms, sys: 15.1 ms, total: 142 ms
Wall time: 73.7 ms

In [134]: res[:3]
Out[134]: 0    0.387242
1   -0.969343
2   -0.863159
dtype: float64

In [135]: %time res = np.sum(df.values, axis=1)  ❺
CPU times: user 49.3 ms, sys: 2.36 ms, total: 51.7 ms
Wall time: 26.9 ms

In [136]: res[:3]
Out[136]: array([ 0.3872424 , -0.96934273, -0.86315944])
```

❶ Working with the columns (`Series` objects) directly is the fastest approach.

❷ This calculates the sums by calling the `sum()` method on the `DataFrame` object.

❸ This calculates the sums by calling the `sum()` method on the `ndarray` object.

❹ This calculates the sums by using the function `np.sum()` on the `DataFrame` object.

❺ This calculates the sums by using the function `np.sum()` on the `ndarray` object.

Finally, two more options which are based on the methods `eval()` and `apply()`, respectively:[1]

```
In [137]: %time res = df.eval('x + y')  ❶
          CPU times: user 25.5 ms, sys: 17.7 ms, total: 43.2 ms
          Wall time: 22.5 ms

In [138]: res[:3]
Out[138]: 0    0.387242
          1   -0.969343
          2   -0.863159
          dtype: float64

In [139]: %time res = df.apply(lambda row: row['x'] + row['y'], axis=1)  ❷
          CPU times: user 19.6 s, sys: 83.3 ms, total: 19.7 s
          Wall time: 19.9 s

In [140]: res[:3]
Out[140]: 0    0.387242
          1   -0.969343
          2   -0.863159
          dtype: float64
```

❶ `eval()` is a method dedicated to evaluation of (complex) numerical expressions; columns can be directly addressed.

❷ The slowest option is to use the `apply()` method row-by-row; this is like looping on the Python level over all rows.

> **CHOOSE WISELY**
>
> `pandas` often provides multiple options to accomplish the same goal. If unsure of which to use, compare the options to verify that the best possible performance is achieved when time is critical. In this simple example, execution times differ by orders of magnitude.

Conclusion

`pandas` is a powerful tool for data analysis and has become the central package in the so-called *PyData* stack. Its `DataFrame` class is particularly suited to working with tabular data of any kind. Most operations on such objects are vectorized, leading not only—as in the `NumPy` case—to concise code but also to

high performance in general. In addition, `pandas` makes working with incomplete data sets convenient (which is not the case with `NumPy`, for instance). `pandas` and the `DataFrame` class will be central in many later chapters of the book, where additional features will be used and introduced when necessary.

Further Reading

`pandas` is an open source project with both online documentation and a PDF version available for download.[2] The website provides links to both, and additional resources:

- *http://pandas.pydata.org/*

As for `NumPy`, recommended references for `pandas` in book form are:

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

---

1 The application of the `eval()` method requires the `numexpr` package to be installed.

2 At the time of this writing, the PDF version has a total of more than 2,500 pages.

Part III. Financial Data Science

This part of the book is about basic techniques, approaches, and packages for financial data science. Many topics (such as visualization) and many packages (such as `scikit-learn`) are fundamental for data science with Python. In that sense, this part equips the quants and financial analysts with the Python tools they need to become *financial data scientists*.

Like in Part II, the chapters are organized according to topics such that they can each be used as a reference for the topic of interest:

- Chapter 7 discusses static and interactive visualization with `matplotlib` and `plotly`.
- Chapter 8 is about handling financial time series data with `pandas`.
- Chapter 9 focuses on getting input/output (I/O) operations right and fast.
- Chapter 10 is all about making Python code fast.
- Chapter 11 focuses on frequently required mathematical tools in finance.
- Chapter 12 looks at using Python to implement methods from stochastics.
- Chapter 13 is about statistical and machine learning approaches.

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.

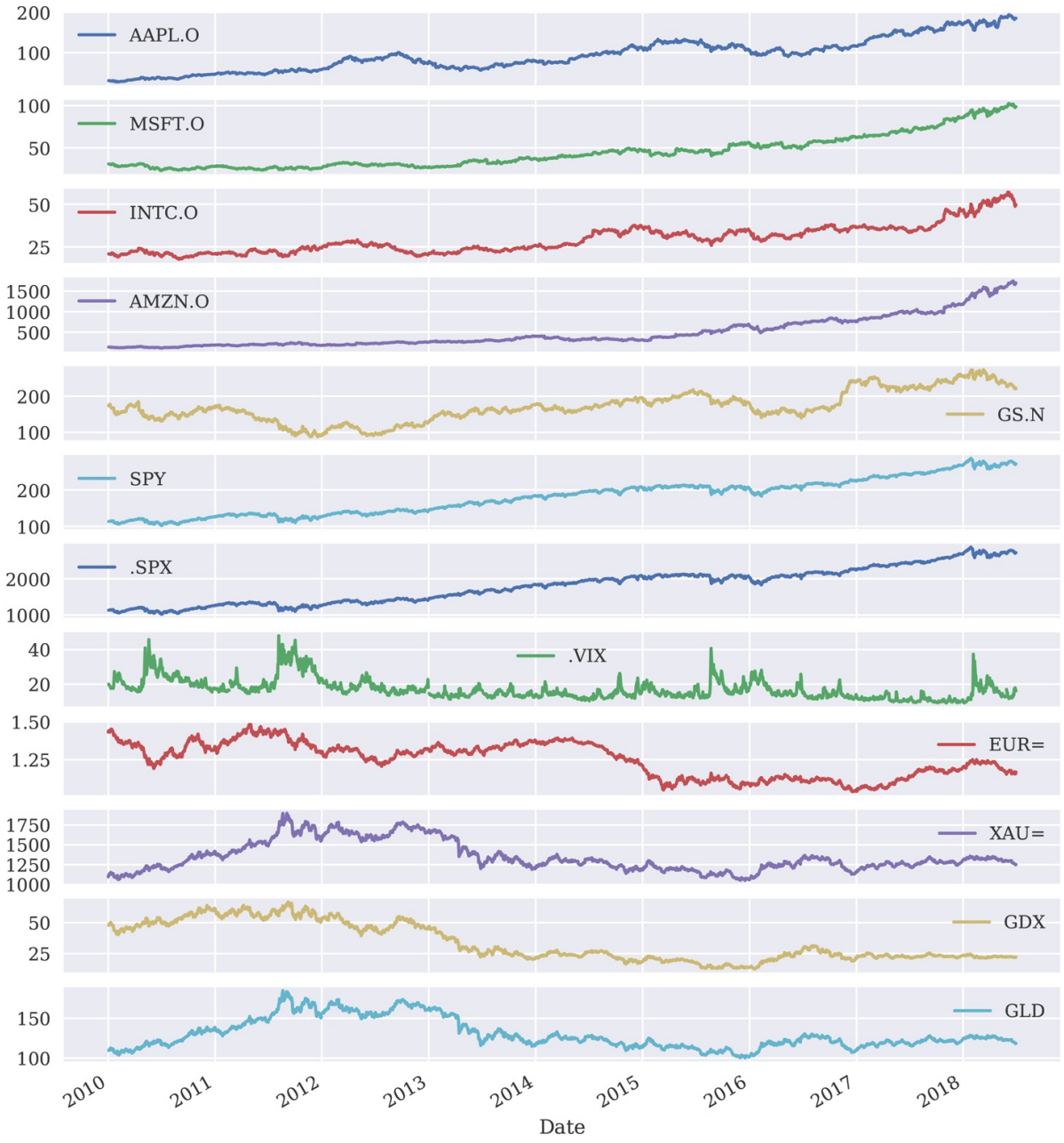

Figure 8-1. Financial time series data as line plots

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.

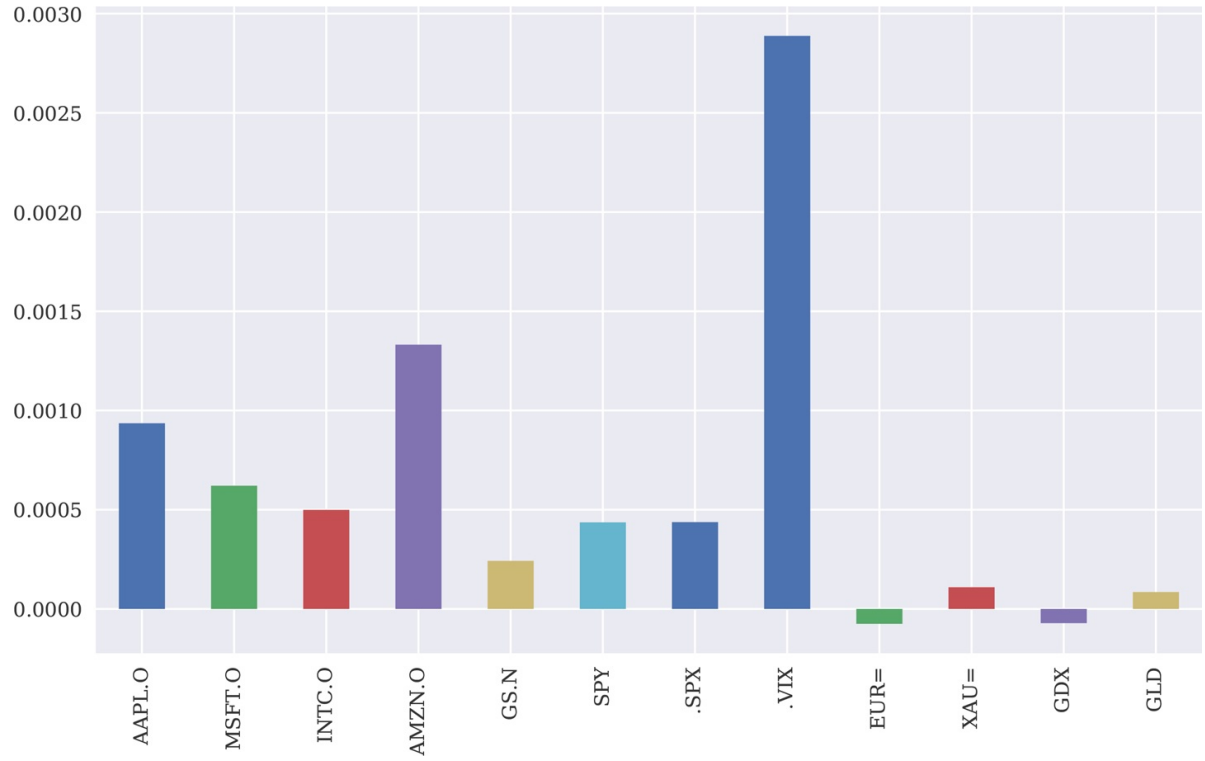

Figure 8-2. Mean values of percentage changes as bar plot

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.

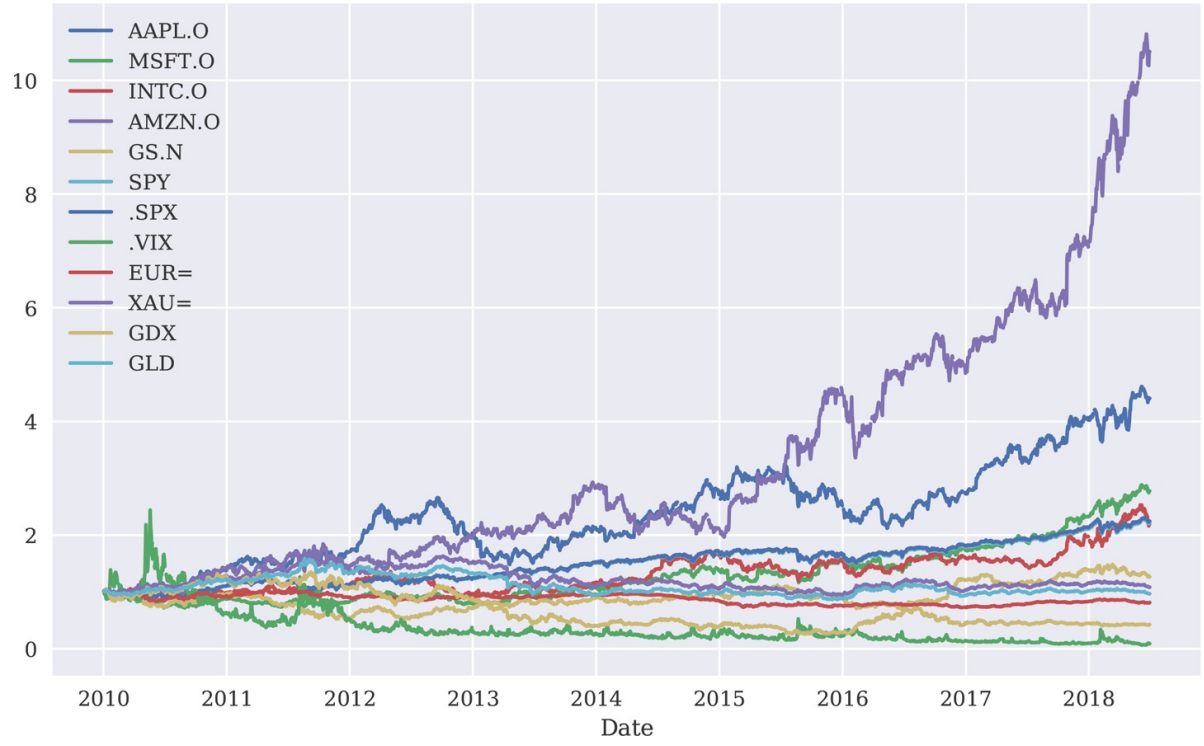

Figure 8-3. Cumulative log returns over time

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.

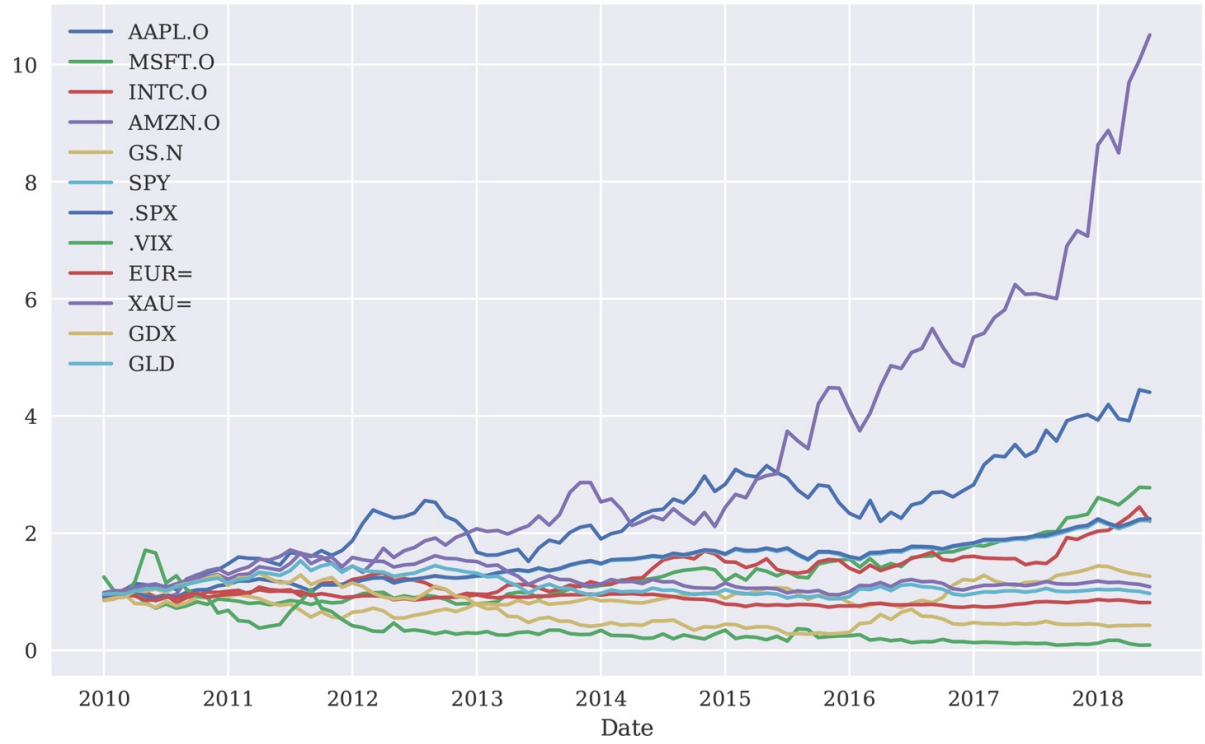

Figure 8-4. Resampled cumulative log returns over time (monthly)

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.

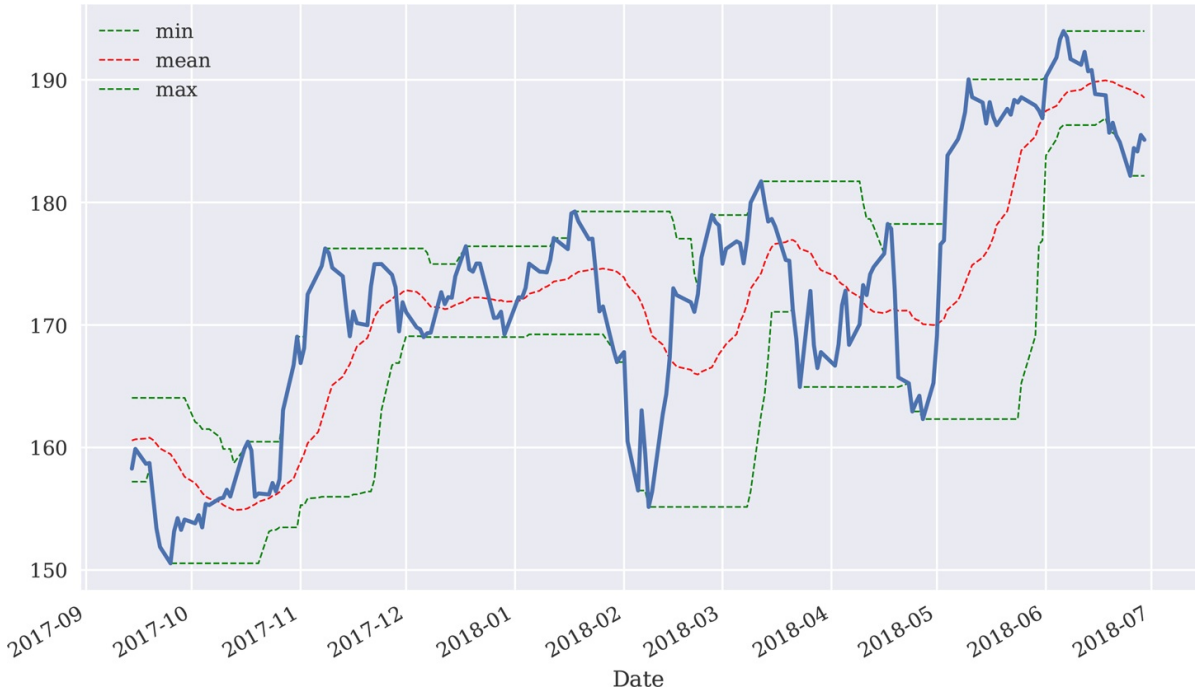

Figure 8-5. Rolling statistics for minimum, mean, maximum values

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`).

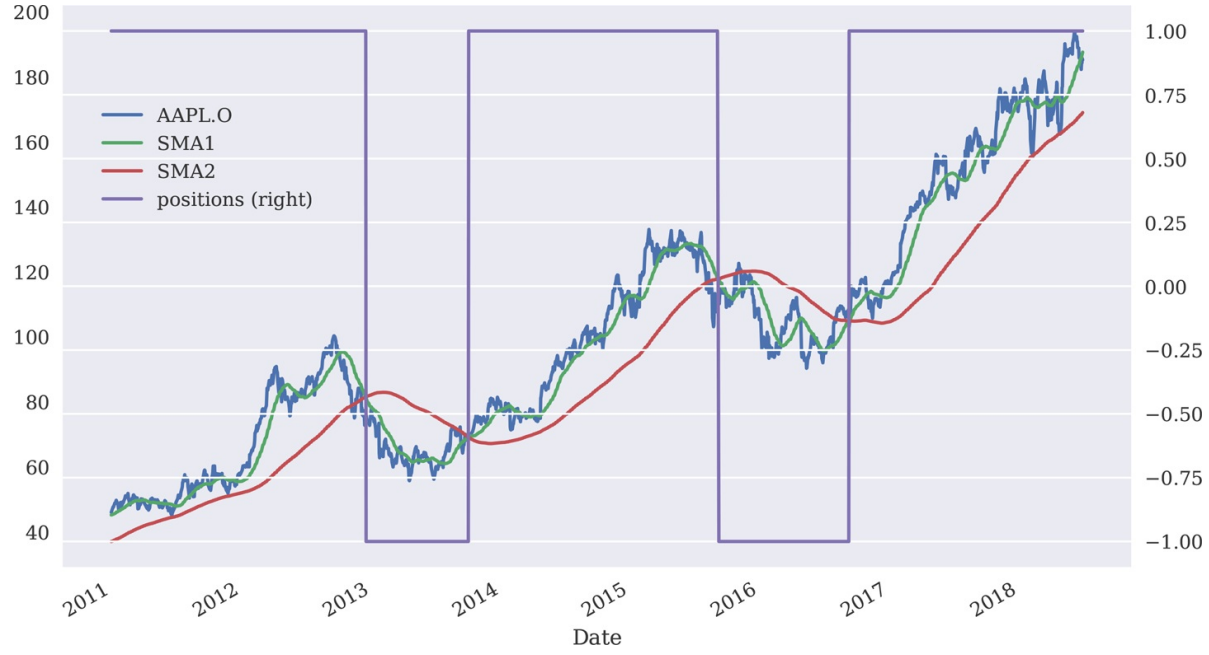

Figure 8-7. Apple stock price, two simple moving averages and positions

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.

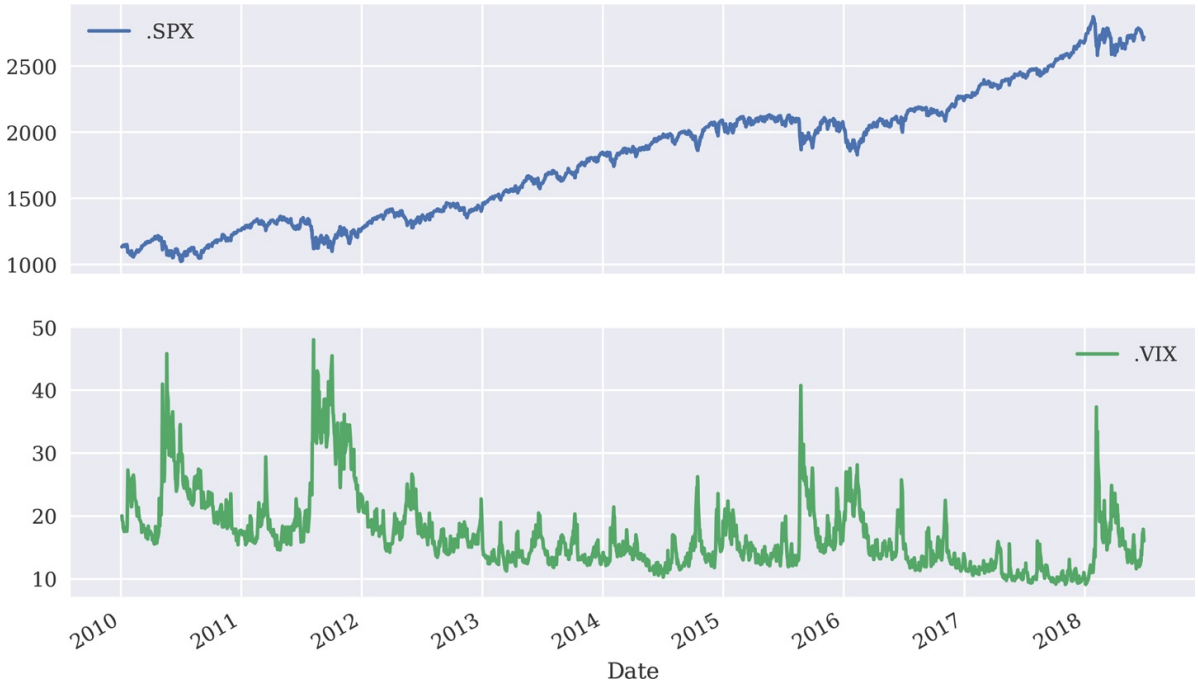

Figure 8-8. S&P 500 and VIX time series data (different subplots)

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`.

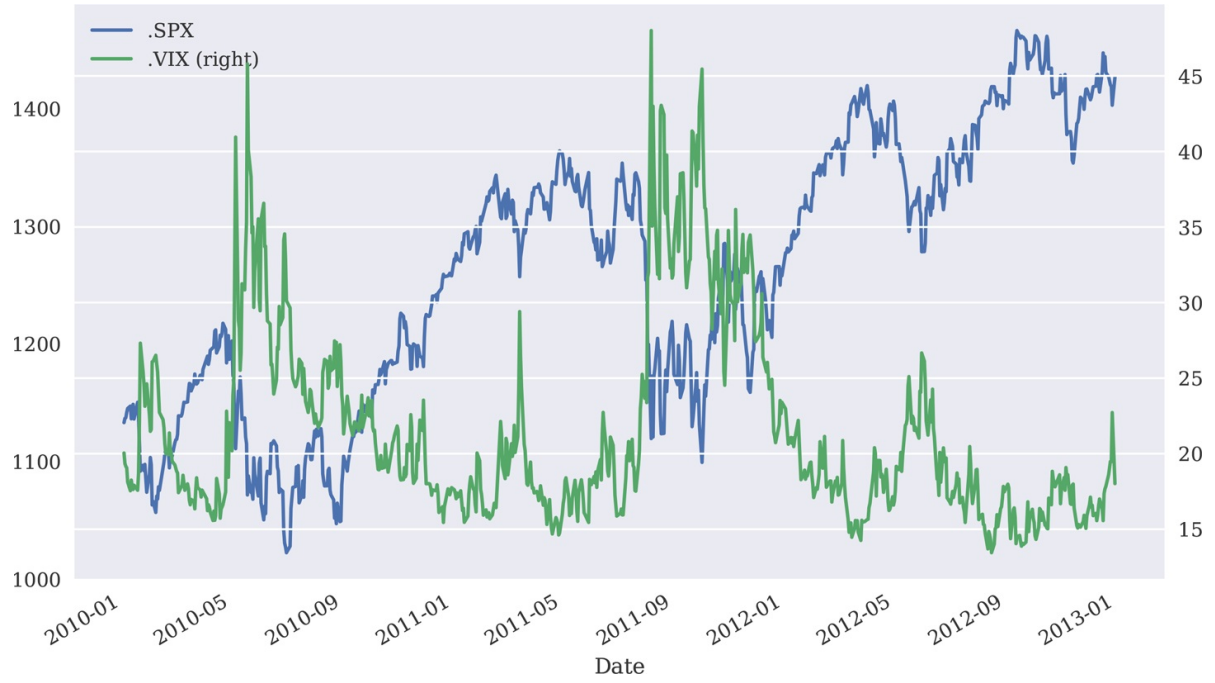

Figure 8-9. S&P 500 and VIX time series data (same plot)

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.

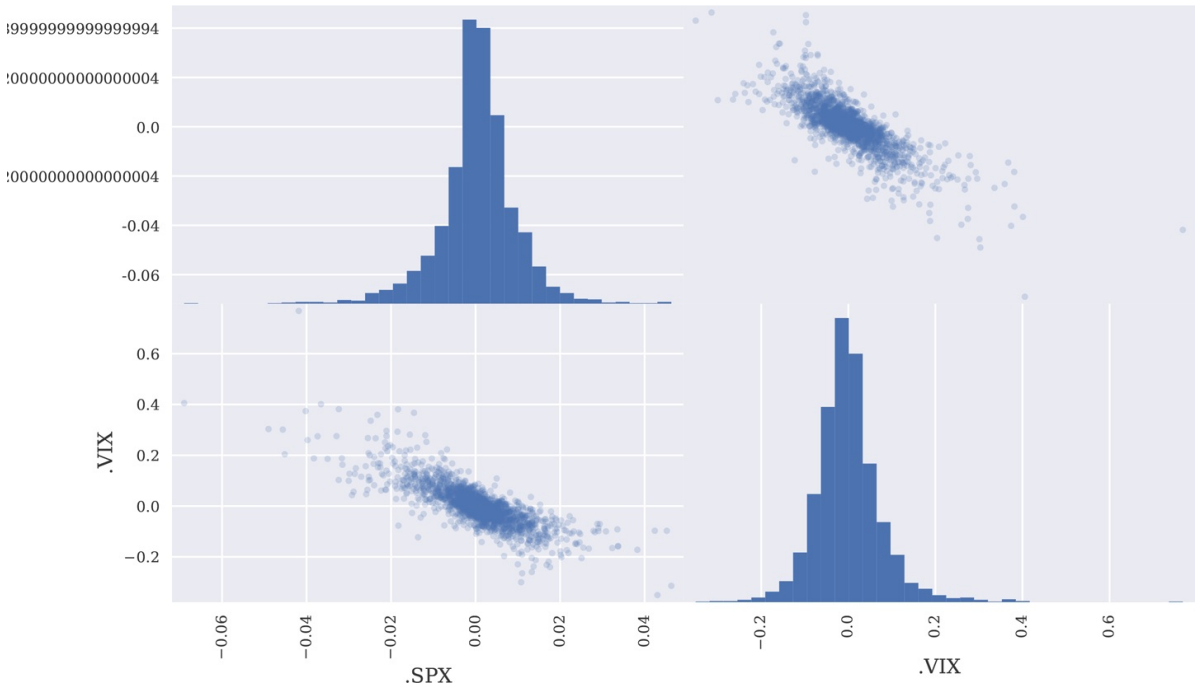

Figure 8-11. Log returns of the S&P 500 and VIX as a scatter matrix

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.

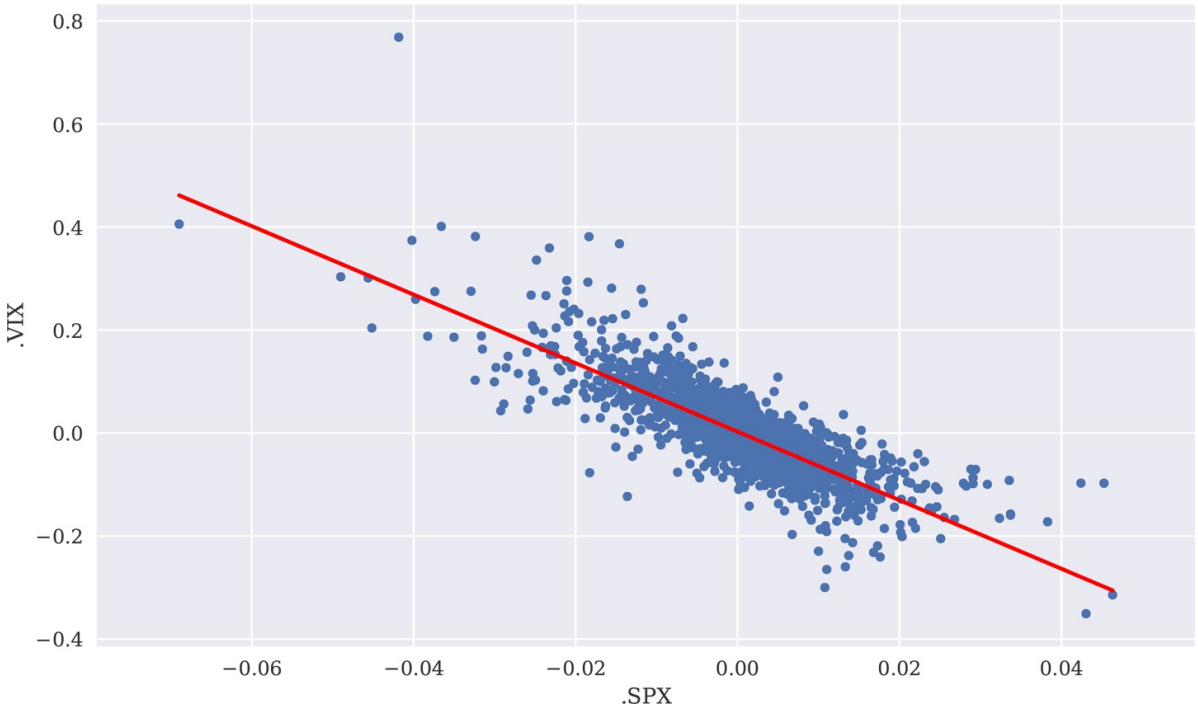

Figure 8-12. Log returns of the S&P 500 and VIX as a scatter matrix

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.

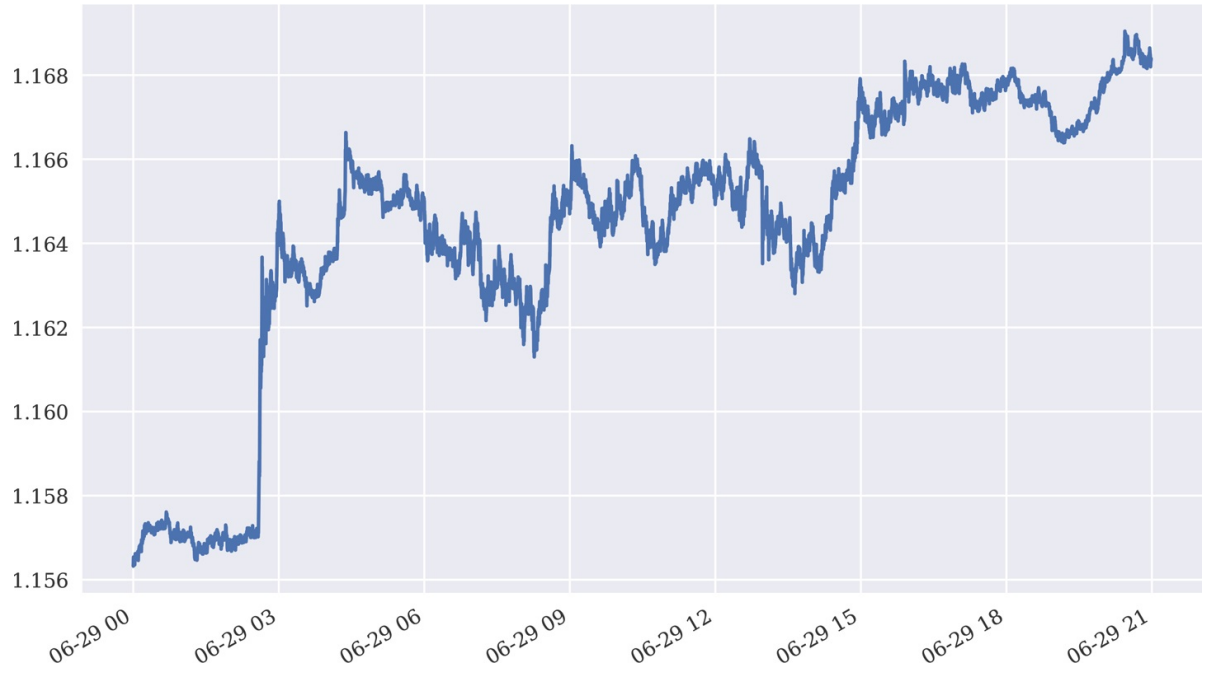

Figure 8-14. Tick data for EUR/USD exchange rate

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.

Chapter 10. Performance Python

> *Don't lower your expectations to meet your performance. Raise your level of performance to meet your expectations.*
>
> —Ralph Marston

It is a long-lived prejudice that Python per se is a relatively slow programming language and not appropriate to implement computationally demanding tasks in finance. Beyond the fact that Python is an interpreted language, the reasoning is usually along the following lines: Python is slow when it comes to loops; loops are often required to implement financial algorithms; therefore Python is too slow for financial algorithm implementation. Another line of reasoning is: other (compiled) programming languages are fast at executing loops (such as C or C++); loops are often required for financial algorithms; therefore these (compiled) programming languages are well suited for finance and financial algorithm implementation.

Admittedly, it is possible to write proper Python code that executes rather slowly—perhaps too slowly for many application areas. This chapter is about approaches to speed up typical tasks and algorithms often encountered in a financial context. It shows that with a judicious use of data structures, choosing the right implementation idioms and paradigms, as well as using the right performance packages, Python is able to compete even with compiled programming languages. This is due to, among other factors, getting compiled itself.

To this end, this chapter introduces different approaches to speed up code:

Vectorization

: Making use of Python's vectorization capabilities is one approach already used extensively in previous chapters.

Dynamic compiling

: Using the `Numba` package allows one to dynamically compile pure Python code using LLVM technology.

Static compiling

: `Cython` is not only a Python package but a hybrid language that combines Python and C; it allows one, for instance, to use static type declarations and to statically compile such adjusted code.

Multiprocessing

: The `multiprocessing` module of Python allows for easy and simple parallelization of code execution.

This chapter addresses the following topics:

"Loops"

: This section addresses Python loops and how to speed them up.

"Algorithms"

: This section is concerned with standard mathematical algorithms that are often used for performance benchmarks, such as Fibonacci number generation.

"Binomial Trees"

: The binomial option pricing model is a widely used financial model that allows for an interesting case study about a more involved financial algorithm.

"Monte Carlo Simulation"

: Similarly, Monte Carlo simulation is widely used in financial practice for pricing and risk management. It is computationally demanding and has long been considered the domain of such languages as C or C++.

"Recursive pandas Algorithm"

: This section addresses the speedup of a recursive algorithm based on financial time series data. In particular, it presents different implementations for an algorithm calculating an exponentially weighted moving average (EWMA).

Loops

This section tackles the Python loop issue. The task is rather simple: a function shall be written that draws a certain “large” number of random numbers and returns the average of the values. The execution time is of interest, which can be estimated by the magic functions `%time` and `%timeit`.

Python

Let’s get started “slowly”—forgive the pun. In pure Python, such a function might look like `average_py()`:

```
In [1]: import random

In [2]: def average_py(n):
            s = 0  ❶
            for i in range(n):
                s += random.random()  ❷
            return s / n  ❸

In [3]: n = 10000000  ❹

In [4]: %time average_py(n)  ❺
        CPU times: user 1.82 s, sys: 10.4 ms, total: 1.83 s
        Wall time: 1.93 s

Out[4]: 0.5000590124747943

In [5]: %timeit average_py(n)  ❻
        1.31 s ± 159 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)

In [6]: %time sum([random.random() for _ in range(n)]) / n  ❼
        CPU times: user 1.55 s, sys: 188 ms, total: 1.74 s
        Wall time: 1.74 s

Out[6]: 0.49987031710661173
```

❶ Initializes the variable value for `s`.

❷ Adds the uniformly distributed random values from the interval (0, 1) to `s`.

❸ Returns the average value (mean).

❹ Defines the number of iterations for the loop.

❺ Times the function once.

❻ Times the function multiple times for a more reliable estimate.

❼ Uses a `list` comprehension instead of the function.

This sets the benchmark for the other approaches to follow.

NumPy

The strength of `NumPy` lies in its vectorization capabilities. Formally, loops vanish on the Python level; the looping takes place one level deeper based on optimized and compiled routines provided by `NumPy`.[1] The function `average_np()` makes use of this approach:

```
In [7]: import numpy as np

In [8]: def average_np(n):
            s = np.random.random(n)  ❶
            return s.mean()  ❷

In [9]: %time average_np(n)
        CPU times: user 180 ms, sys: 43.2 ms, total: 223 ms
        Wall time: 224 ms

Out[9]: 0.49988861556468317

In [10]: %timeit average_np(n)
         128 ms ± 2.01 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)

In [11]: s = np.random.random(n)
         s.nbytes  ❸
Out[11]: 80000000
```

❶ Draws the random numbers “all at once” (no Python loop).

❷ Returns the average value (mean).

❸ Number of bytes used for the created `ndarray` object.

The speedup is considerable, reaching almost a factor of 10 or an order of magnitude. However, the price that must be paid is significantly higher memory usage. This is due to the fact that `NumPy` attains speed by preallocating data that can be processed in the compiled layer. As a consquence, there is no way, given this approach, to work with “streamed” data. This increased memory usage might even be prohibitively large depending on the algorithm or problem at

hand.

> **VECTORIZATION AND MEMORY**
>
> It is tempting to write vectorized code with NumPy whenever possible due to the concise syntax and speed improvements typically observed. However, these benefits often come at the price of a much higher memory footprint.

Numba

Numba is a package that allows the *dynamic compiling* of pure Python code by the use of LLVM. The application in a simple case, like the one at hand, is surprisingly straightforward and the dynamically compiled function `average_nb()` can be called directly from Python:

```
In [12]: import numba

In [13]: average_nb = numba.jit(average_py)  ❶

In [14]: %time average_nb(n)  ❷
         CPU times: user 204 ms, sys: 34.3 ms, total: 239 ms
         Wall time: 278 ms

Out[14]: 0.4998865391283664

In [15]: %time average_nb(n)  ❸
         CPU times: user 80.9 ms, sys: 457 µs, total: 81.3 ms
         Wall time: 81.7 ms

Out[15]: 0.5001357454250273

In [16]: %timeit average_nb(n)  ❸
         75.5 ms ± 1.95 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

❶ This creates the Numba function.

❷ The compiling happens during runtime, leading to some overhead.

❸ From the second execution (with the same input data types), the execution is faster.

The combination of pure Python with Numba beats the NumPy version *and* preserves the memory efficiency of the original loop-based implementation. It is

also obvious that the application of `Numba` in such simple cases comes with hardly any programming overhead.

> **NO FREE LUNCH**
>
> The application of `Numba` sometimes seems like magic when one compares the performance of the Python code to the compiled version, especially given its ease of use. However, there are many use cases for which `Numba` is not suited and for which performance gains are hardly observed or even impossible to achieve.

Cython

`Cython` allows one to *statically compile* Python code. However, the application is not as simple as with `Numba` since the code generally needs to be changed to see significant speed improvements. To begin with, consider the `Cython` function `average_cy1()`, which introduces static type declarations for the used variables:

```
In [17]: %load_ext Cython

In [18]: %%cython -a
         import random  ❶
         def average_cy1(int n):  ❷
             cdef int i  ❷
             cdef float s = 0  ❷
             for i in range(n):
                 s += random.random()
             return s / n
Out[18]: <IPython.core.display.HTML object>

In [19]: %time average_cy1(n)
         CPU times: user 695 ms, sys: 4.31 ms, total: 699 ms
         Wall time: 711 ms

Out[19]: 0.49997106194496155

In [20]: %timeit average_cy1(n)
         752 ms ± 91.1 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

❶ Imports the `random` module within the `Cython` context.

❷ Adds static type declarations for the variables `n`, `i`, and `s`.

Some speedup is observed, but not even close to that achieved by, for example,

the NumPy version. A bit more Cython optimization is necessary to beat even the Numba version:

```
In [21]: %%cython
         from libc.stdlib cimport rand  ❶
         cdef extern from 'limits.h':  ❷
             int INT_MAX  ❷
         cdef int i
         cdef float rn
         for i in range(5):
             rn = rand() / INT_MAX  ❸
             print(rn)
         0.6792964339256287
         0.934692919254303
         0.3835020661354065
         0.5194163918495178
         0.8309653401374817

In [22]: %%cython -a
         from libc.stdlib cimport rand  ❶
         cdef extern from 'limits.h':  ❷
             int INT_MAX  ❷
         def average_cy2(int n):
             cdef int i
             cdef float s = 0
             for i in range(n):
                 s += rand() / INT_MAX  ❸
             return s / n
Out[22]: <IPython.core.display.HTML object>

In [23]: %time average_cy2(n)
         CPU times: user 78.5 ms, sys: 422 µs, total: 79 ms
         Wall time: 79.1 ms

Out[23]: 0.500017523765564

In [24]: %timeit average_cy2(n)
         65.4 ms ± 706 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

❶ Imports a random number generator from C.

❷ Imports a constant value for the scaling of the random numbers.

❸ Adds uniformly distributed random numbers from the interval (0, 1), after scaling.

This further optimized Cython version, `average_cy2()`, is now a bit faster than the Numba version. However, the effort has also been a bit larger. Compared to

the NumPy version, Cython also preserves the memory efficiency of the original loop-based implementation.

> **CYTHON = PYTHON + C**
>
> Cython allows developers to tweak code for performance as much as possible or as little as sensible—starting with a pure Python version, for instance, and adding more and more elements from C to the code. The compilation step itself can also be parameterized to further optimize the compiled version.

Algorithms

This section applies the performance-enhancing techniques from the previous section to some well-known problems and algorithms from mathematics. These algorithms are regularly used for performance benchmarks.

Prime Numbers

Prime numbers play an important role not only in theoretical mathematics but also in many applied computer science disciplines, such as encryption. A *prime number* is a positive natural number greater than 1 that is only divisible without remainder by 1 and itself. There are no other factors. While it is difficult to find larger prime numbers due to their rarity, it is easy to prove that a number is not prime. The only thing that is needed is a factor other than 1 that divides the number without a remainder.

Python

There are a number of algorithmic implementations available to test if numbers are prime. The following is a Python version that is not yet optimal from an algorithmic point of view but is already quite efficient. The execution time for the larger prime p2, however, is long:

```
In [25]: def is_prime(I):
             if I % 2 == 0: return False  ❶
             for i in range(3, int(I ** 0.5) + 1, 2):  ❷
                 if I % i == 0: return False  ❸
             return True  ❹
```

```
In [26]: n = int(1e8 + 3)  ❺
         n
Out[26]: 100000003

In [27]: %time is_prime(n)
         CPU times: user 35 µs, sys: 0 ns, total: 35 µs
         Wall time: 39.1 µs

Out[27]: False

In [28]: p1 = int(1e8 + 7)  ❺
         p1
Out[28]: 100000007

In [29]: %time is_prime(p1)
         CPU times: user 776 µs, sys: 1 µs, total: 777 µs
         Wall time: 787 µs

Out[29]: True

In [30]: p2 = 100109100129162907  ❻

In [31]: p2.bit_length()  ❻
Out[31]: 57

In [32]: %time is_prime(p2)
         CPU times: user 22.6 s, sys: 44.7 ms, total: 22.6 s
         Wall time: 22.7 s

Out[32]: True
```

❶ If the number is even, `False` is returned immediately.
❷ The loop starts at 3 and goes until the square root of `I` plus 1 with step size 2.
❸ As soon as a factor is identified the function returns `False`.
❹ If no factor is found, `True` is returned.
❺ Relatively small non-prime and prime numbers.
❻ A larger prime number which requires longer execution times.

Numba

The loop structure of the algorithm in the function `is_prime()` lends itself well to being dynamically compiled with `Numba`. The overhead again is minimal but the speedup considerable:

```
In [33]: is_prime_nb = numba.jit(is_prime)

In [34]: %time is_prime_nb(n)  ❶
         CPU times: user 87.5 ms, sys: 7.91 ms, total: 95.4 ms
         Wall time: 93.7 ms

Out[34]: False

In [35]: %time is_prime_nb(n)  ❷
         CPU times: user 9 µs, sys: 1e+03 ns, total: 10 µs
         Wall time: 13.6 µs

Out[35]: False

In [36]: %time is_prime_nb(p1)
         CPU times: user 26 µs, sys: 0 ns, total: 26 µs
         Wall time: 31 µs

Out[36]: True

In [37]: %time is_prime_nb(p2)  ❸
         CPU times: user 1.72 s, sys: 9.7 ms, total: 1.73 s
         Wall time: 1.74 s

Out[37]: True
```

❶ The first call of `is_prime_nb()` involves the compiling overhead.
❷ From the second call, the speedup becomes fully visible.
❸ The speedup for the larger prime is about an order of magnitude.

Cython

The application of `Cython` is straightforward as well. A plain `Cython` version without type declarations already speeds up the code significantly:

```
In [38]: %%cython
         def is_prime_cy1(I):
             if I % 2 == 0: return False
             for i in range(3, int(I ** 0.5) + 1, 2):
                 if I % i == 0: return False
             return True

In [39]: %timeit is_prime(p1)
         394 µs ± 14.7 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)

In [40]: %timeit is_prime_cy1(p1)
```

```
          243 µs ± 6.58 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

However, real improvements only materialize with the static type declarations. The `Cython` version then even is slightly faster than the `Numba` one:

```
In [41]: %%cython
         def is_prime_cy2(long I):  ❶
             cdef long i  ❶
             if I % 2 == 0: return False
             for i in range(3, int(I ** 0.5) + 1, 2):
                 if I % i == 0: return False
             return True

In [42]: %timeit is_prime_cy2(p1)
         87.6 µs ± 27.7 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)

In [43]: %time is_prime_nb(p2)
         CPU times: user 1.68 s, sys: 9.73 ms, total: 1.69 s
         Wall time: 1.7 s

Out[43]: True

In [44]: %time is_prime_cy2(p2)
         CPU times: user 1.66 s, sys: 9.47 ms, total: 1.67 s
         Wall time: 1.68 s

Out[44]: True
```

❶ Static type declarations for the two variables `I` and `i`.

Multiprocessing

So far, all the optimization efforts have focused on the sequential code execution. In particular with prime numbers, there might be a need to check multiple numbers at the same time. To this end, the `multiprocessing` module can help speed up the code execution further. It allows one to spawn multiple Python processes that run in parallel. The application is straightforward in the simple case at hand. First, an `mp.Pool` object is set up with multiple processes. Second, the function to be executed is *mapped* to the prime numbers to be checked:

```
In [45]: import multiprocessing as mp
```

```
In [46]: pool = mp.Pool(processes=4)  ❶

In [47]: %time pool.map(is_prime, 10 * [p1])  ❷
         CPU times: user 1.52 ms, sys: 2.09 ms, total: 3.61 ms
         Wall time: 9.73 ms

Out[47]: [True, True, True, True, True, True, True, True, True, True]

In [48]: %time pool.map(is_prime_nb, 10 * [p2])  ❷
         CPU times: user 13.9 ms, sys: 4.8 ms, total: 18.7 ms
         Wall time: 10.4 s

Out[48]: [True, True, True, True, True, True, True, True, True, True]

In [49]: %time pool.map(is_prime_cy2, 10 * [p2])  ❷
         CPU times: user 9.8 ms, sys: 3.22 ms, total: 13 ms
         Wall time: 9.51 s

Out[49]: [True, True, True, True, True, True, True, True, True, True]
```

❶ The `mp.Pool` object is instantiated with multiple processes.

❷ Then the respective function is mapped to a `list` object with prime numbers.

The observed speedup is significant. The Python function `is_prime()` takes more than 20 seconds for the larger prime number `p2`. Both the `is_prime_nb()` and the `is_prime_cy2()` functions take less than 10 seconds for 10 times the prime number `p2` when executed in parallel with four processes.

> **PARALLEL PROCESSING**
>
> Parallel processing should be considered whenever different problems of the same type need to be solved. The effect can be huge when powerful hardware is available with many cores and sufficient working memory. `multiprocessing` is one easy-to-use module from the standard library.

Fibonacci Numbers

Fibonacci numbers and sequences can be derived based on a simple algorithm. Start with two ones: 1, 1. From the third number, the next Fibonacci number is derived as the sum of the two preceding ones: 1, 1, 2, 3, 5, 8, 13, 21, …. This section analyzes two different implementations, a recursive one and an iterative

❹ They suffer from an overflow issue due to the restriction to 64-bit `int` objects.

❺ Imports the special 128-bit `int` object type and uses it.

❻ The `Cython` version `fib_it_cy2()` now is faster *and* correct.

The Number Pi

The final algorithm analyzed in this section is a Monte Carlo simulation–based algorithm to derive digits for the number pi (π).[2] The basic idea relies on the fact that the area *A* of a circle is given by $A = \pi r^2$. Therefore, $\pi = \frac{A}{r^2}$. For a unit circle with radius $r = 1$, it holds that $\pi = A$. The idea of the algorithm is to simulate random points with coordinate values (x, y), with $x, y \in [-1, 1]$. The area of an origin-centered square with side length of 2 is exactly 4. The area of the origin-centered unit circle is a fraction of the area of such a square. This fraction can be estimated by Monte Carlo simulation: count all the points in the square, then count all the points in the circle, and divide the number of points in the circle by the number of points in the square. The following example demonstrates (see Figure 10-1):

```
In [76]: import random
         import numpy as np
         from pylab import mpl, plt
         plt.style.use('seaborn')
         mpl.rcParams['font.family'] = 'serif'
         %matplotlib inline

In [77]: rn = [(random.random() * 2 - 1, random.random() * 2 - 1)
               for _ in range(500)]

In [78]: rn = np.array(rn)
         rn[:5]
Out[78]: array([[ 0.45583018, -0.27676067],
                [-0.70120038,  0.15196888],
                [ 0.07224045,  0.90147321],
                [-0.17450337, -0.47660912],
                [ 0.94896746, -0.31511879]])

In [79]: fig = plt.figure(figsize=(7, 7))
         ax = fig.add_subplot(1, 1, 1)
         circ = plt.Circle((0, 0), radius=1, edgecolor='g', lw=2.0,
                           facecolor='None')  ❶
         box = plt.Rectangle((-1, -1), 2, 2, edgecolor='b', alpha=0.3)  ❷
```

```
ax.add_patch(circ)  ❶
ax.add_patch(box)  ❷
plt.plot(rn[:, 0], rn[:, 1], 'r.')  ❸
plt.ylim(-1.1, 1.1)
plt.xlim(-1.1, 1.1)
```

❶ Draws the unit circle.

❷ Draws the square with side length of 2.

❸ Draws the uniformly distributed random dots.

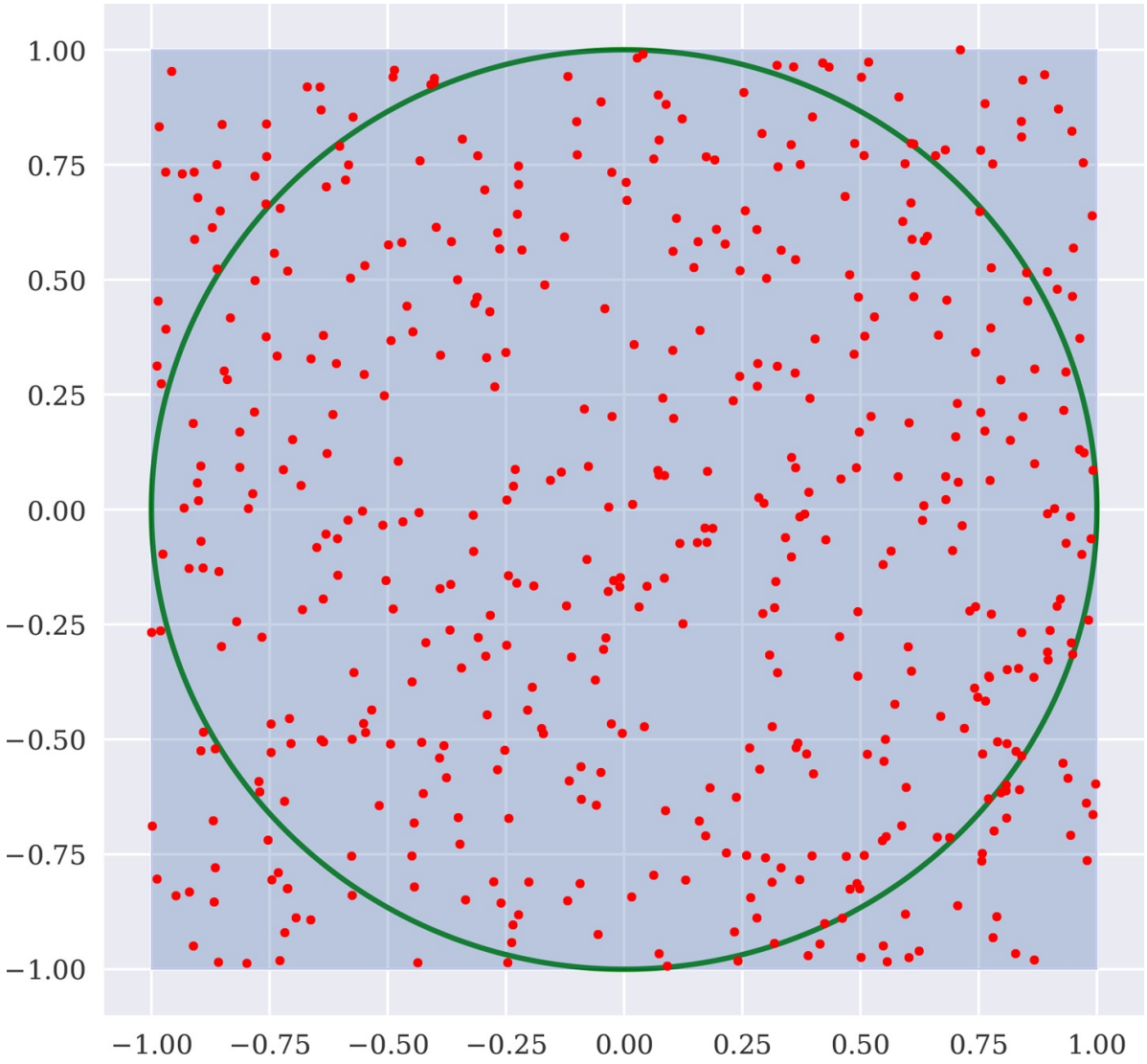

Figure 10-1. Unit circle and square with side length 2 with uniformly distributed random points

A `NumPy` implementation of this algorithm is rather concise but also memory-intensive. Total execution time given the parameterization is about one second:

```
In [80]: n = int(1e7)

In [81]: %time rn = np.random.random((n, 2)) * 2 - 1
         CPU times: user 450 ms, sys: 87.9 ms, total: 538 ms
         Wall time: 573 ms

In [82]: rn.nbytes
Out[82]: 160000000

In [83]: %time distance = np.sqrt((rn ** 2).sum(axis=1))  ❶
         distance[:8].round(3)
         CPU times: user 537 ms, sys: 198 ms, total: 736 ms
         Wall time: 651 ms

Out[83]: array([1.181, 1.061, 0.669, 1.206, 0.799, 0.579, 0.694, 0.941])

In [84]: %time frac = (distance <= 1.0).sum() / len(distance)  ❷
         CPU times: user 47.9 ms, sys: 6.77 ms, total: 54.7 ms
         Wall time: 28 ms

In [85]: pi_mcs = frac * 4  ❸
         pi_mcs  ❸
Out[85]: 3.1413396
```

❶ The distance of the points from the origin (Euclidean norm).

❷ The fraction of those points on the circle relative to all points.

❸ This accounts for the square area of 4 for the estimation of the circle area and therewith of π.

`mcs_pi_py()` is a Python function using a `for` loop and implementing the Monte Carlo simulation in a memory-efficient manner. Note that the random numbers are not scaled in this case. The execution time is longer than with the `NumPy` version, but the `Numba` version is faster than `NumPy` in this case:

```
In [86]: def mcs_pi_py(n):
             circle = 0
             for _ in range(n):
                 x, y = random.random(), random.random()
                 if (x ** 2 + y ** 2) ** 0.5 <= 1:
                     circle += 1
             return (4 * circle) / n

In [87]: %time mcs_pi_py(n)
         CPU times: user 5.47 s, sys: 23 ms, total: 5.49 s
         Wall time: 5.43 s
```

```
Out[87]: 3.1418964

In [88]: mcs_pi_nb = numba.jit(mcs_pi_py)

In [89]: %time mcs_pi_nb(n)
         CPU times: user 319 ms, sys: 6.36 ms, total: 326 ms
         Wall time: 326 ms

Out[89]: 3.1422012

In [90]: %time mcs_pi_nb(n)
         CPU times: user 284 ms, sys: 3.92 ms, total: 288 ms
         Wall time: 291 ms

Out[90]: 3.142066
```

A plain `Cython` version with static type declarations only does not perform that much faster than the Python version. However, relying again on the random number generation capabilities of C further speeds up the calculation considerably:

```
In [91]: %%cython -a
         import random
         def mcs_pi_cy1(int n):
             cdef int i, circle = 0
             cdef float x, y
             for i in range(n):
                 x, y = random.random(), random.random()
                 if (x ** 2 + y ** 2) ** 0.5 <= 1:
                     circle += 1
             return (4 * circle) / n
Out[91]: <IPython.core.display.HTML object>

In [92]: %time mcs_pi_cy1(n)
         CPU times: user 1.15 s, sys: 8.24 ms, total: 1.16 s
         Wall time: 1.16 s

Out[92]: 3.1417132

In [93]: %%cython -a
         from libc.stdlib cimport rand
         cdef extern from 'limits.h':
             int INT_MAX
         def mcs_pi_cy2(int n):
             cdef int i, circle = 0
             cdef float x, y
             for i in range(n):
```

```
                x, y = rand() / INT_MAX, rand() / INT_MAX
                if (x ** 2 + y ** 2) ** 0.5 <= 1:
                    circle += 1
            return (4 * circle) / n
```

```
Out[93]: <IPython.core.display.HTML object>
```

```
In [94]: %time mcs_pi_cy2(n)
         CPU times: user 170 ms, sys: 1.45 ms, total: 172 ms
         Wall time: 172 ms
```

```
Out[94]: 3.1419388
```

ALGORITHM TYPES

The algorithms analyzed in this section might not be directly related to financial algorithms. However, the advantage is that they are simple and easy to understand. In addition, typical algorithmic problems encountered in a financial context can be discussed within this simplified context.

Binomial Trees

A popular numerical method to value options is the binomial option pricing model pioneered by Cox, Ross, and Rubinstein (1979). This method relies on representing the possible future evolution of an asset by a (recombining) tree. In this model, as in the Black-Scholes-Merton (1973) setup, there is a *risky asset*, an index or stock, and a *riskless asset*, a bond. The relevant time interval from today until the maturity of the option is divided in general into equidistant subintervals of length Δt. Given an index level at time s of S_s, the index level at $t = s + \Delta t$ is given by $S_t = S_s \cdot m$, where m is chosen randomly from $\{u, d\}$ with $0 < d < e^{r\Delta t} < u = e^{\sigma\sqrt{\Delta t}}$ as well as $u = \frac{1}{d}$. r is the constant, riskless short rate.

Python

The code that follows presents a Python implementation that creates a recombining tree based on some fixed numerical parameters for the model:

```
In [95]: import math
```

MULTIPROCESSING STRATEGIES

In finance, there are many algorithms that are useful for parallelization. Some of these even allow the application of different strategies to parallelize the code. Monte Carlo simulation is a good example in that multiple simulations can easily be executed in parallel, either on a single machine or on multiple machines, and that the algorithm itself allows a single simulation to be distributed over multiple processes.

Recursive pandas Algorithm

This section addresses a somewhat special topic which is, however, an important one in financial analytics: the implementation of recursive functions on financial time series data stored in a `pandas` `DataFrame` object. While `pandas` allows for sophisticated vectorized operations on `DataFrame` objects, certain recursive algorithms are hard or impossible to vectorize, leaving the financial analyst with slowly executed Python loops on `DataFrame` objects. The examples that follow implement what is called the *exponentially weighted moving average* (EWMA) in a simple form.

The EWMA for a financial time series $S_t, t \in \{0, \cdots, T\}$, is given by Equation 10-4.

Equation 10-4. Exponentially weighted moving average (EWMA)

$$\begin{aligned} EWMA_0 &= S_0 \\ EWMA_t &= \alpha \cdot S_t + (1 - \alpha) \cdot EWMA_{t-1}, t \in \{1, \cdots, T\} \end{aligned}$$

Although simple in nature and straightforward to implement, such an algorithm might lead to rather slow code.

Python

Consider first the Python version that iterates over the `DatetimeIndex` of a `DataFrame` object containing financial time series data for a single financial instrument (see Chapter 8). Figure 10-3 visualizes the financial time series and the EWMA time series:

```
In [148]: import pandas as pd
```

```
In [149]: sym = 'SPY'

In [150]: data = pd.DataFrame(pd.read_csv('../../source/tr_eikon_eod_data.csv',
                                          index_col=0, parse_dates=True)[sym]).dropna()

In [151]: alpha = 0.25

In [152]: data['EWMA'] = data[sym]  ❶

In [153]: %%time
          for t in zip(data.index, data.index[1:]):
              data.loc[t[1], 'EWMA'] = (alpha * data.loc[t[1], sym] +
                                        (1 - alpha) * data.loc[t[0], 'EWMA'])  ❷
          CPU times: user 588 ms, sys: 16.4 ms, total: 605 ms
          Wall time: 591 ms

In [154]: data.head()
Out[154]:                 SPY        EWMA
          Date
          2010-01-04  113.33  113.330000
          2010-01-05  113.63  113.405000
          2010-01-06  113.71  113.481250
          2010-01-07  114.19  113.658438
          2010-01-08  114.57  113.886328

In [155]: data[data.index > '2017-1-1'].plot(figsize=(10, 6));
```

❶ Initializes the `EWMA` column.

❷ Implements the algorithm based on a Python loop.

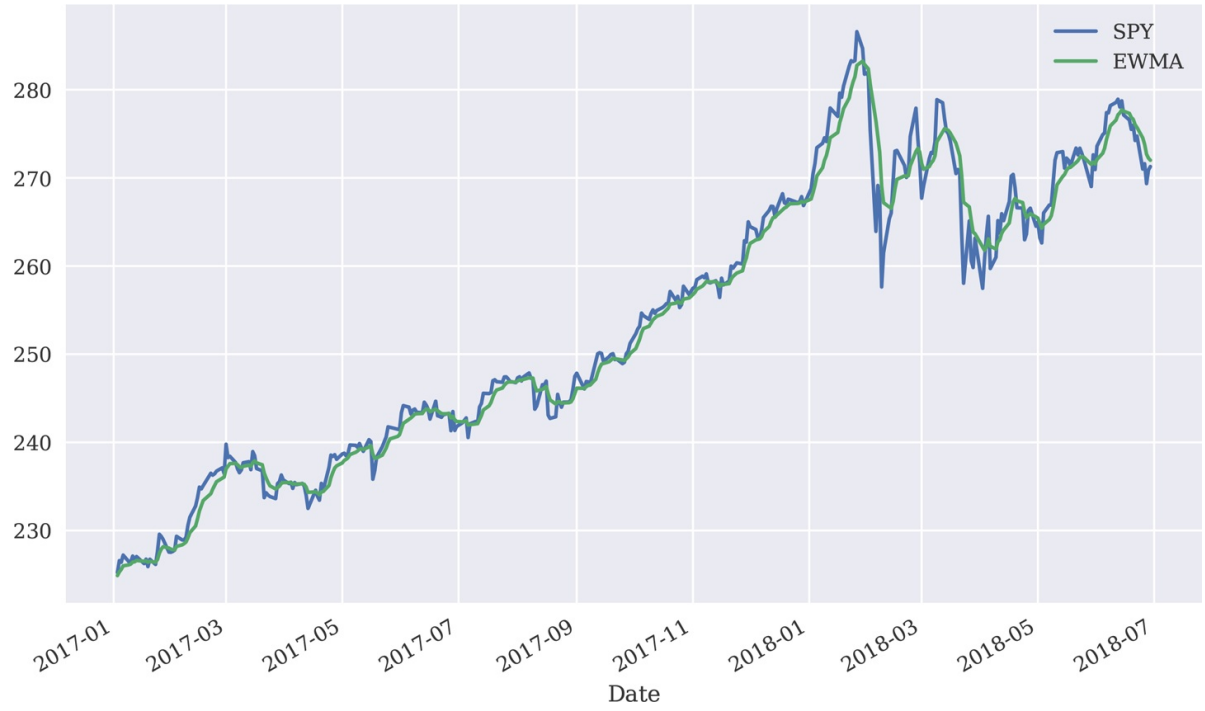

Figure 10-3. Financial time series with EWMA

Now consider more general Python function `ewma_py()`. It can be applied directly on the column or the raw financial times series data in the form of an `ndarray` object:

```
In [156]: def ewma_py(x, alpha):
              y = np.zeros_like(x)
              y[0] = x[0]
              for i in range(1, len(x)):
                  y[i] = alpha * x[i] + (1-alpha) * y[i-1]
              return y

In [157]: %time data['EWMA_PY'] = ewma_py(data[sym], alpha)  ❶
          CPU times: user 33.1 ms, sys: 1.22 ms, total: 34.3 ms
          Wall time: 33.9 ms

In [158]: %time data['EWMA_PY'] = ewma_py(data[sym].values, alpha)  ❷
          CPU times: user 1.61 ms, sys: 44 µs, total: 1.65 ms
          Wall time: 1.62 ms
```

❶ Applies the function to the `Series` object directly (i.e., the column).

❷ Applies the function to the `ndarray` object containing the raw data.

This approach already speeds up the code execution considerably—by a factor of from about 20 to more than 100.

Numba

The very structure of this algorithm promises further speedups when applying `Numba`. And indeed, when the function `ewma_nb()` is applied to the `ndarray` version of the data, the speedup is again by an order of magnitude:

```
In [159]: ewma_nb = numba.jit(ewma_py)

In [160]: %time data['EWMA_NB'] = ewma_nb(data[sym], alpha)  ❶
          CPU times: user 269 ms, sys: 11.4 ms, total: 280 ms
          Wall time: 294 ms

In [161]: %timeit data['EWMA_NB'] = ewma_nb(data[sym], alpha)  ❶
          30.9 ms ± 1.21 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)

In [162]: %time data['EWMA_NB'] = ewma_nb(data[sym].values, alpha)  ❷
          CPU times: user 94.1 ms, sys: 3.78 ms, total: 97.9 ms
          Wall time: 97.6 ms

In [163]: %timeit data['EWMA_NB'] = ewma_nb(data[sym].values, alpha)  ❷
          134 µs ± 12.5 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

❶ Applies the function to the `Series` object directly (i.e., the column).

❷ Applies the function to the `ndarray` object containing the raw data.

Cython

The `Cython` version, `ewma_cy()`, also achieves considerable speed improvements but it is not as fast as the `Numba` version in this case:

```
In [164]: %%cython
          import numpy as np
          cimport cython
          @cython.boundscheck(False)
          @cython.wraparound(False)
          def ewma_cy(double[:] x, float alpha):
              cdef int i
              cdef double[:] y = np.empty_like(x)
              y[0] = x[0]
              for i in range(1, len(x)):
                  y[i] = alpha * x[i] + (1 - alpha) * y[i - 1]
              return y

In [165]: %time data['EWMA_CY'] = ewma_cy(data[sym].values, alpha)
```

```
            CPU times: user 2.98 ms, sys: 1.41 ms, total: 4.4 ms
            Wall time: 5.96 ms

In [166]: %timeit data['EWMA_CY'] = ewma_cy(data[sym].values, alpha)
            1.29 ms ± 194 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

This final example illustrates again that there are in general multiple options to implement (nonstandard) algorithms. All options might lead to exactly the same results, while also showing considerably different performance characteristics. The execution times in this example range from 0.1 ms to 500 ms—a factor of 5,000 times.

BEST VERSUS FIRST-BEST

It is easy in general to translate algorithms to the Python programming language. However, it is equally easy to implement algorithms in a way that is unnecessarily slow given the menu of performance options available. For interactive financial analytics, a *first-best* solution—i.e., one that does the trick but which might not be the fastest possible nor the most memory-efficient one—might be perfectly fine. For financial applications in production, one should strive to implement the *best* solution, even though this might involve a bit more research and some formal benchmarking.

Conclusion

The Python ecosystem provides a number of ways to improve the performance of code:

Idioms and paradigms

: Some Python paradigms and idioms might be more performant than others, given a specific problem; in many cases, for instance, vectorization is a paradigm that not only leads to more concise code but also to higher speeds (sometimes at the cost of a larger memory footprint).

Packages

: There are a wealth of packages available for different types of problems, and using a package adapted to the problem can often lead to much higher performance; good examples are `NumPy` with the `ndarray` class and `pandas` with the `DataFrame` class.

Compiling

: Powerful packages for speeding up financial algorithms are `Numba` and `Cython` for the dynamic and static compilation of Python code.

Parallelization

: Some Python packages, such as `multiprocessing`, allow for the easy parallelization of Python code; the examples in this chapter only use parallelization on a single machine but the Python ecosystem also offers technologies for multi-machine (cluster) parallelization.

A major benefit of the performance approaches presented in this chapter is that they are in general easily implementable, meaning that the additional effort required is regularly low. In other words, performance improvements often are low-hanging fruit given the performance packages available as of today.

Further Resources

For all the performance packages introduced in this chapter, there are helpful web resources available:

- *http://cython.org* is the home of the `Cython` package and compiler project.
- The documentation for the `multiprocessing` module is found at *https://docs.python.org/3/library/multiprocessing.html*.
- Information on `Numba` can be found at *http://github.com/numba/numba* and *https://numba.pydata.org*.

For references in book form, see the following:

- Gorelick, Misha, and Ian Ozsvald (2014). *High Performance Python*. Sebastopol, CA: O'Reilly.
- Smith, Kurt (2015). *Cython*. Sebastopol, CA: O'Reilly.

Chapter 11. Mathematical Tools

The mathematicians are the priests of the modern world.

—Bill Gaede

Since the arrival of the so-called Rocket Scientists on Wall Street in the 1980s and 1990s, finance has evolved into a discipline of applied mathematics. While early research papers in finance came with lots of text and few mathematical expressions and equations, current ones are mainly comprised of mathematical expressions and equations with some explanatory text around.

This chapter introduces some useful mathematical tools for finance, without providing a detailed background for each of them. There are many useful books available on this topic, so this chapter focuses on how to use the tools and techniques with Python. In particular, it covers:

"Approximation"

Regression and interpolation are among the most often used numerical techniques in finance.

"Convex Optimization"

A number of financial disciplines need tools for convex optimization (for instance, derivatives analytics when it comes to model calibration).

"Integration"

In particular, the valuation of financial (derivative) assets often boils down to the evaluation of integrals.

"Symbolic Computation"

Python provides with `SymPy` a powerful package for symbolic mathematics, for example, to solve (systems of) equations.

Approximation

To begin with, the usual imports:

```
In [1]: import numpy as np
        from pylab import plt, mpl

In [2]: plt.style.use('seaborn')
        mpl.rcParams['font.family'] = 'serif'
        %matplotlib inline
```

Throughout this section, the main example function is the following, which is comprised of a trigonometric term and a linear term:

```
In [3]: def f(x):
            return np.sin(x) + 0.5 * x
```

The main focus is the approximation of this function over a given interval by *regression* and *interpolation* techniques. First, a plot of the function to get a better view of what exactly the approximation shall achieve. The interval of interest shall be [–2π, 2π]. Figure 11-1 displays the function over the fixed interval defined via the `np.linspace()` function. The function `create_plot()` is a helper function to create the same type of plot required multiple times in this chapter:

```
In [4]: def create_plot(x, y, styles, labels, axlabels):
            plt.figure(figsize=(10, 6))
            for i in range(len(x)):
                plt.plot(x[i], y[i], styles[i], label=labels[i])
                plt.xlabel(axlabels[0])
                plt.ylabel(axlabels[1])
            plt.legend(loc=0)

In [5]: x = np.linspace(-2 * np.pi, 2 * np.pi, 50)  ❶

In [6]: create_plot([x], [f(x)], ['b'], ['f(x)'], ['x', 'f(x)'])
```

❶ The *x* values used for the plotting and the calculations.

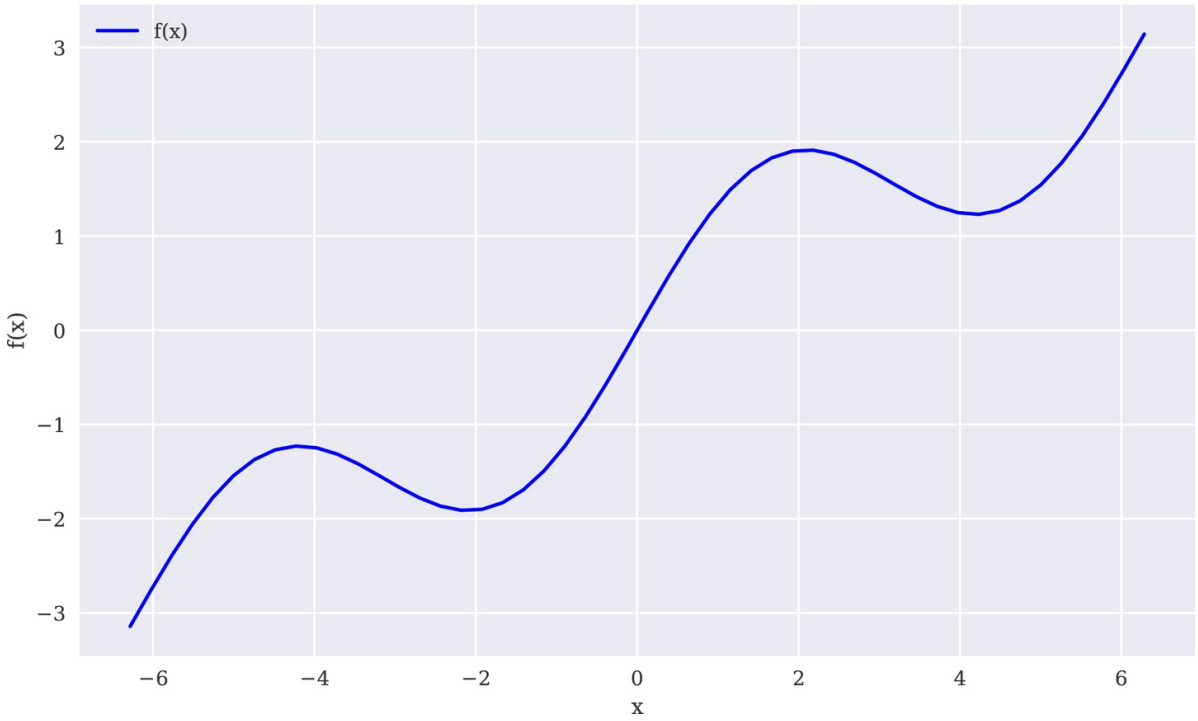

Figure 11-1. Example function plot

Regression

Regression is a rather efficient tool when it comes to function approximation. It is not only suited to approximating one-dimensional functions but also works well in higher dimensions. The numerical techniques needed to come up with regression results are easily implemented and quickly executed. Basically, the task of regression, given a set of so-called basis functions $b_d, d \in \{1, \cdots, D\}$, is to find optimal parameters $\alpha_1^*, \cdots, \alpha_D^*$ according to Equation 11-1, where $y_i \equiv f(x_i)$ for $i \in \{1, \cdots, I\}$ observation points. The x_i are considered *independent* observations and the y_i *dependent* observations (in a functional or statistical sense).

Equation 11-1. Minimization problem of regression

$$\min_{\alpha_1,\ldots,\alpha_D} \frac{1}{I} \sum_{i=1}^{I} \left(y_i - \sum_{d=1}^{D} \alpha_d \cdot b_d(x_i) \right)^2$$

Monomials as basis functions

One of the simplest cases is to take monomials as basis functions—i.e., $b_1 = 1, b_2 = x, b_3 = x^2, b_4 = x^3, \cdots$. In such a case, `NumPy` has built-in functions for both the determination of the optimal parameters (namely, `np.polyfit()`) and the evaluation of the approximation given a set of input values (namely, `np.polyval()`).

Table 11-1 lists the parameters the `np.polyfit()` function takes. Given the returned optimal regression coefficients `p` from `np.polyfit()`, `np.polyval(p, x)` then returns the regression values for the *x* coordinates.

Table 11-1. Parameters of polyfit() function

Parameter	Description
`x`	*x* coordinates (independent variable values)
`y`	*y* coordinates (dependent variable values)
`deg`	Degree of the fitting polynomial
`full`	If `True`, returns diagnostic information in addition
`w`	Weights to apply to the *y* coordinates
`cov`	If `True`, returns covariance matrix in addition

In typical vectorized fashion, the application of `np.polyfit()` and `np.polyval()` takes on the following form for a linear regression (i.e., for `deg=1`). Given the regression estimates stored in the `ry` array, we can compare the regression result with the original function as presented in Figure 11-2. Of course, a linear regression cannot account for the `sin` part of the example function:

```
In [7]: res = np.polyfit(x, f(x), deg=1, full=True)  ❶

In [8]: res  ❷
Out[8]: (array([ 4.28841952e-01, -1.31499950e-16]),
         array([21.03238686]),
         2,
         array([1., 1.]),
         1.1102230246251565e-14)
```

```
In [9]: ry = np.polyval(res[0], x)  ❸

In [10]: create_plot([x, x], [f(x), ry], ['b', 'r.'],
                     ['f(x)', 'regression'], ['x', 'f(x)'])
```

❶ Linear regression step.

❷ Full results: regression parameters, residuals, effective rank, singular values, and relative condition number.

❸ Evaluation using the regression parameters.

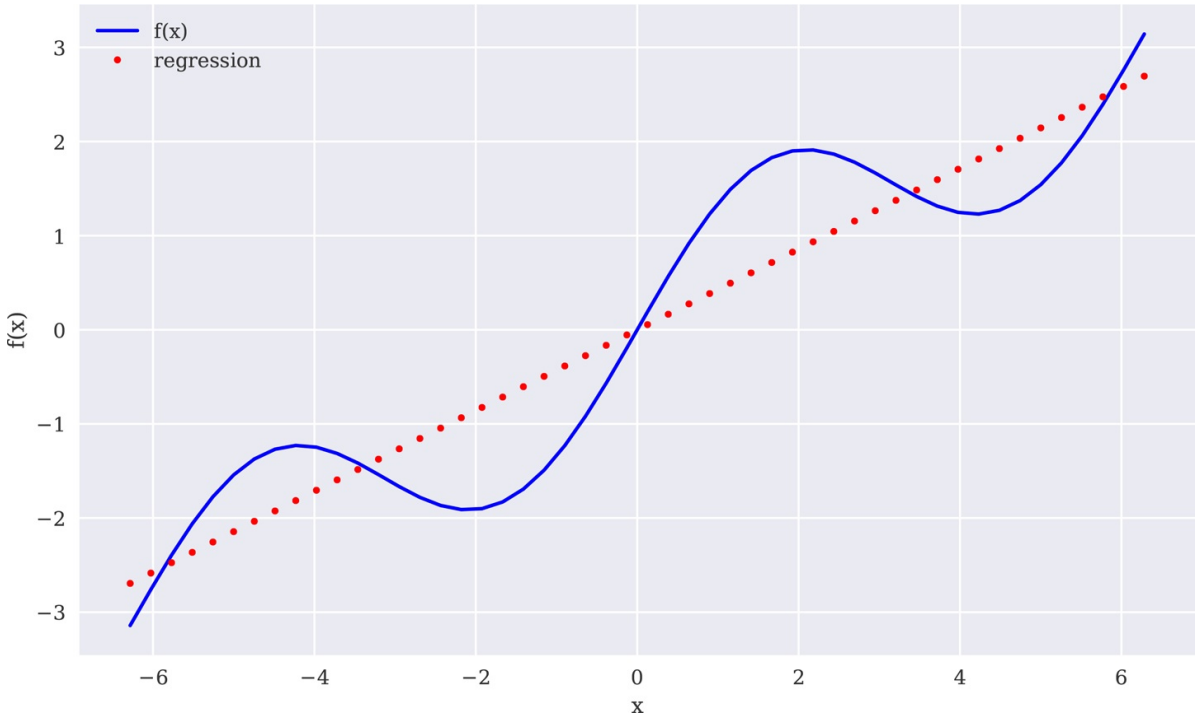

Figure 11-2. Linear regression

To account for the `sin` part of the example function, higher-order monomials are necessary. The next regression attempt takes monomials up to the order of 5 as basis functions. It should not be too surprising that the regression result, as seen in Figure 11-3, now looks much closer to the original function. However, it is still far from being perfect:

```
In [11]: reg = np.polyfit(x, f(x), deg=5)
         ry = np.polyval(reg, x)

In [12]: create_plot([x, x], [f(x), ry], ['b', 'r.'],
                     ['f(x)', 'regression'], ['x', 'f(x)'])
```

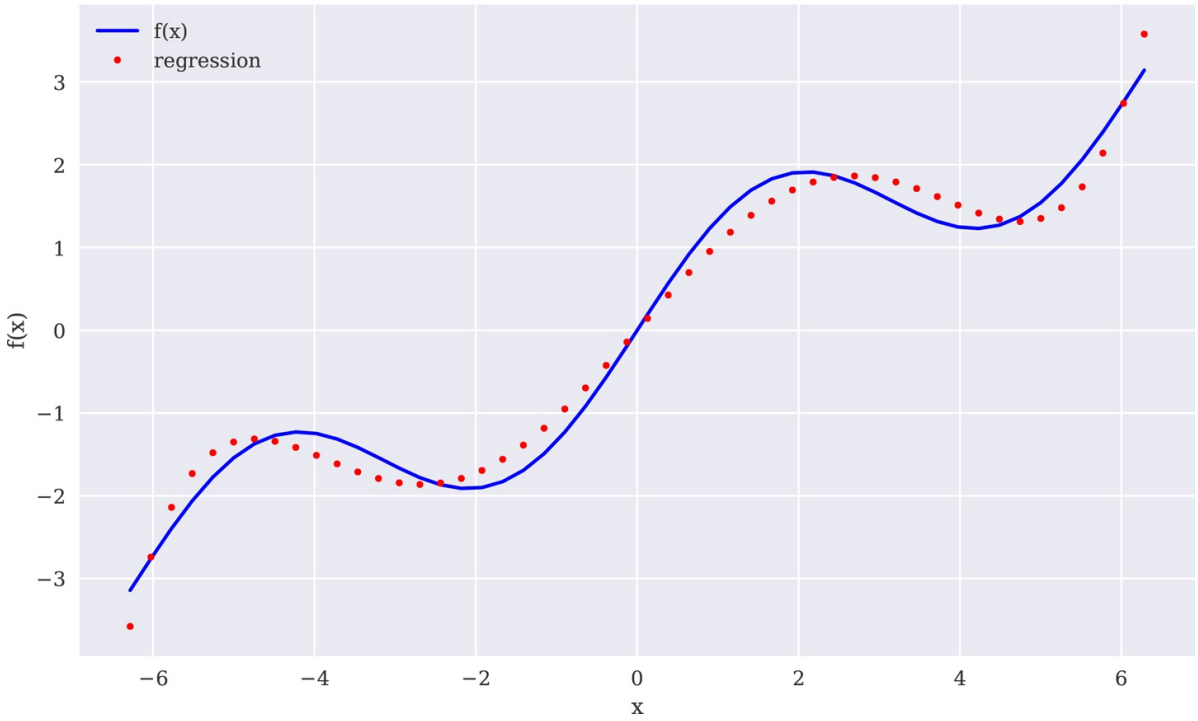

Figure 11-3. Regression with monomials up to order 5

The last attempt takes monomials up to order 7 to approximate the example function. In this case the result, as presented in Figure 11-4, is quite convincing:

```
In [13]: reg = np.polyfit(x, f(x), 7)
         ry = np.polyval(reg, x)

In [14]: np.allclose(f(x), ry)  ❶
Out[14]: False

In [15]: np.mean((f(x) - ry) ** 2)  ❷
Out[15]: 0.0017769134759517689

In [16]: create_plot([x, x], [f(x), ry], ['b', 'r.'],
                     ['f(x)', 'regression'], ['x', 'f(x)'])
```

❶ Checks whether the function and regression values are the same (or at least close).

❷ Calculates the *Mean Squared Error* (MSE) for the regression values given the function values.

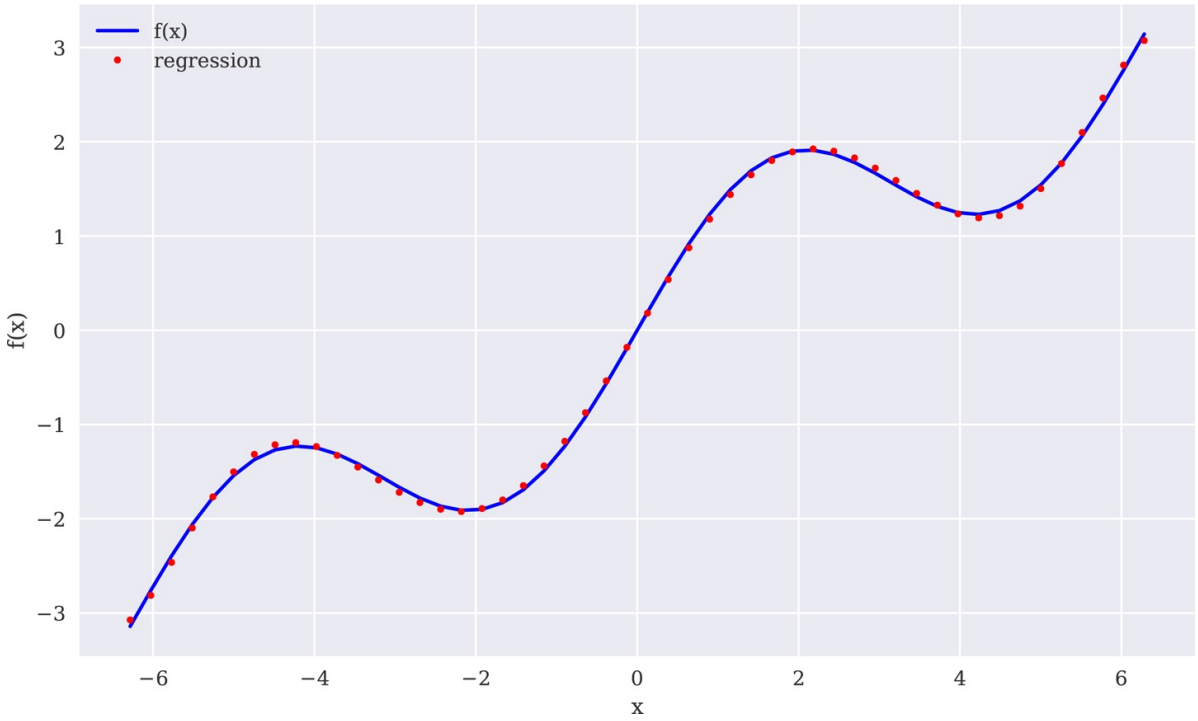

Figure 11-4. Regression with monomials up to order 7

Individual basis functions

In general, one can reach better regression results by choosing better sets of basis functions, e.g., by exploiting knowledge about the function to approximate. In this case, the individual basis functions have to be defined via a matrix approach (i.e., using a `NumPy` `ndarray` object). First, the case with monomials up to order 3 (Figure 11-5). The central function here is `np.linalg.lstsq()`:

```
In [17]: matrix = np.zeros((3 + 1, len(x)))  ❶
         matrix[3, :] = x ** 3  ❷
         matrix[2, :] = x ** 2  ❷
         matrix[1, :] = x  ❷
         matrix[0, :] = 1  ❷

In [18]: reg = np.linalg.lstsq(matrix.T, f(x), rcond=None)[0]  ❸

In [19]: reg.round(4)  ❹
Out[19]: array([ 0.    ,  0.5628, -0.    , -0.0054])

In [20]: ry = np.dot(reg, matrix)  ❺

In [21]: create_plot([x, x], [f(x), ry], ['b', 'r.'],
                     ['f(x)', 'regression'], ['x', 'f(x)'])
```

❶ The `ndarray` object for the basis function values (matrix).
❷ The basis function values from constant to cubic.
❸ The regression step.
❹ The optimal regression parameters.
❺ The regression estimates for the function values.

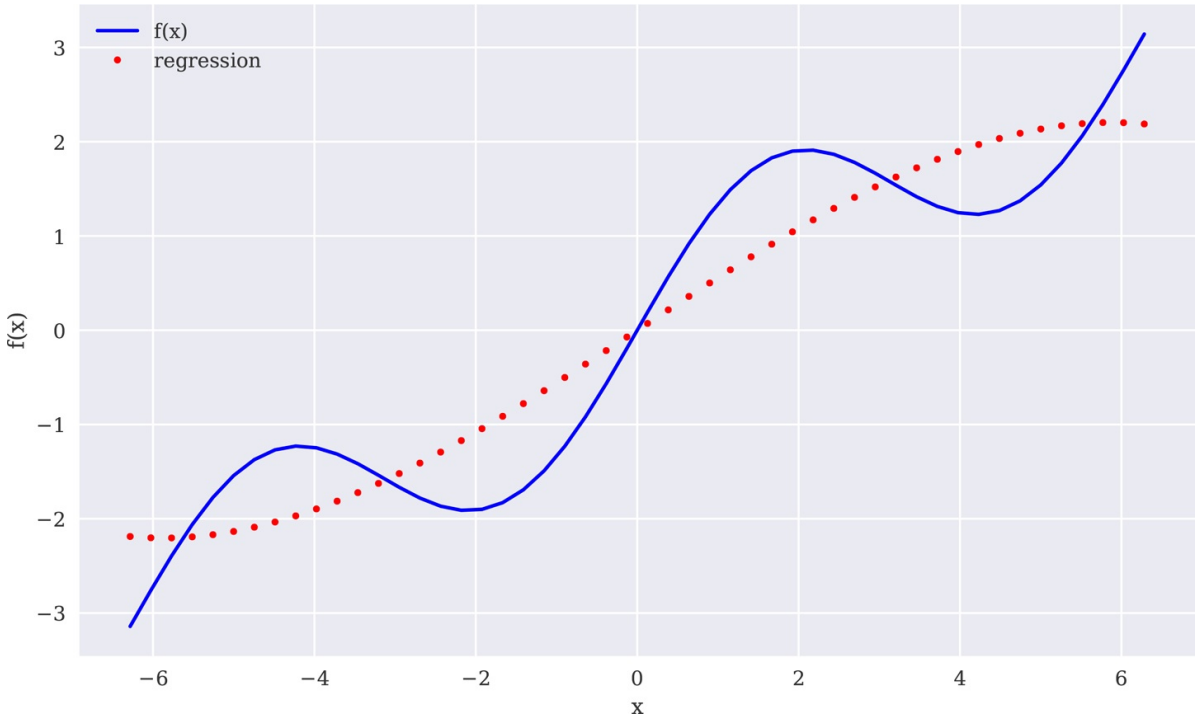

Figure 11-5. Regression with individual basis functions

The result in Figure 11-5 is not as good as expected based on our previous experience with monomials. Using the more general approach allows us to exploit knowledge about the example function—namely that there is a `sin` part in the function. Therefore, it makes sense to include a sine function in the set of basis functions. For simplicity, the highest-order monomial is replaced. The fit now is perfect, as the numbers and Figure 11-6 illustrate:

```
In [22]: matrix[3, :] = np.sin(x)  ❶

In [23]: reg = np.linalg.lstsq(matrix.T, f(x), rcond=None)[0]

In [24]: reg.round(4)  ❷
Out[24]: array([0. , 0.5, 0. , 1. ])

In [25]: ry = np.dot(reg, matrix)
```

```
In [26]: np.allclose(f(x), ry)  ❸
Out[26]: True

In [27]: np.mean((f(x) - ry) ** 2)  ❸
Out[27]: 3.404735992885531e-31

In [28]: create_plot([x, x], [f(x), ry], ['b', 'r.'],
                     ['f(x)', 'regression'], ['x', 'f(x)'])
```

❶ The new basis function exploiting knowledge about the example function.
❷ The optimal regression parameters recover the original parameters.
❸ The regression now leads to a perfect fit.

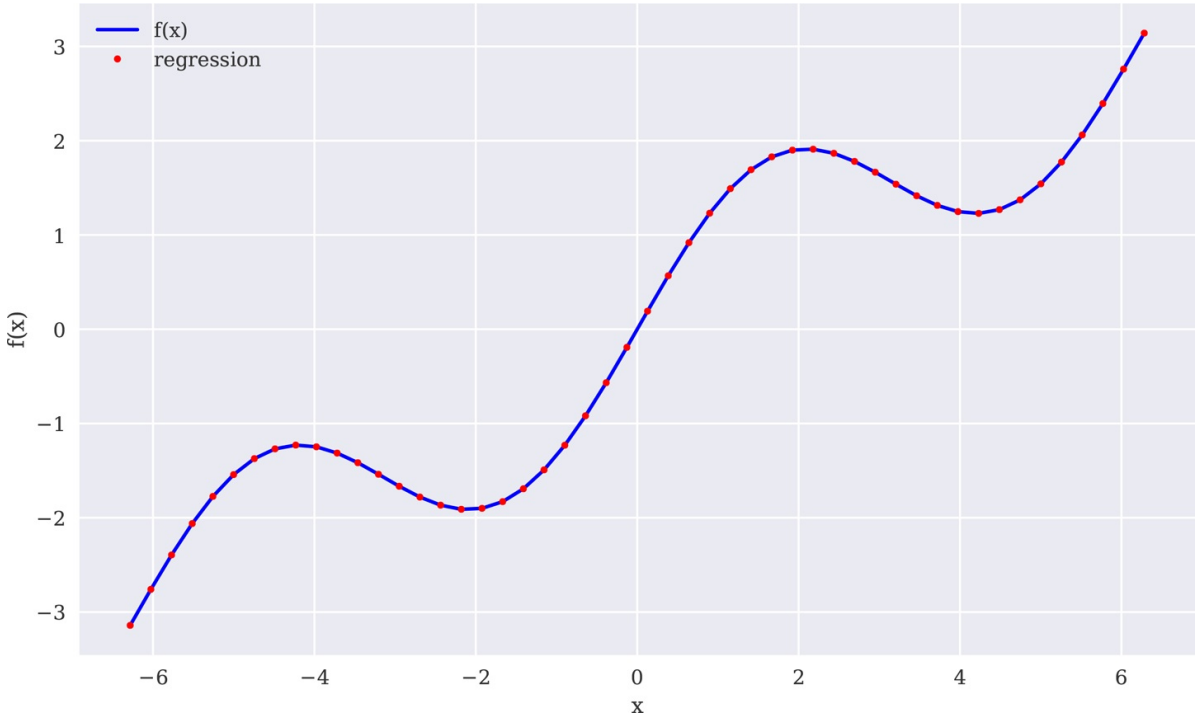

Figure 11-6. Regression with the sine basis function

Noisy data

Regression can cope equally well with *noisy* data, be it data from simulation or from (nonperfect) measurements. To illustrate this point, independent observations with noise and dependent observations with noise are generated. Figure 11-7 reveals that the regression results are closer to the original function than the noisy data points. In a sense, the regression averages out the noise to some extent:

```
In [29]: xn = np.linspace(-2 * np.pi, 2 * np.pi, 50)  ❶
         xn = xn + 0.15 * np.random.standard_normal(len(xn))  ❷
         yn = f(xn) + 0.25 * np.random.standard_normal(len(xn))  ❸

In [30]: reg = np.polyfit(xn, yn, 7)
         ry = np.polyval(reg, xn)

In [31]: create_plot([x, x], [f(x), ry], ['b', 'r.'],
                     ['f(x)', 'regression'], ['x', 'f(x)'])
```

❶ The new deterministic *x* values.
❷ Introducing noise to the *x* values.
❸ Introducing noise to the *y* values.

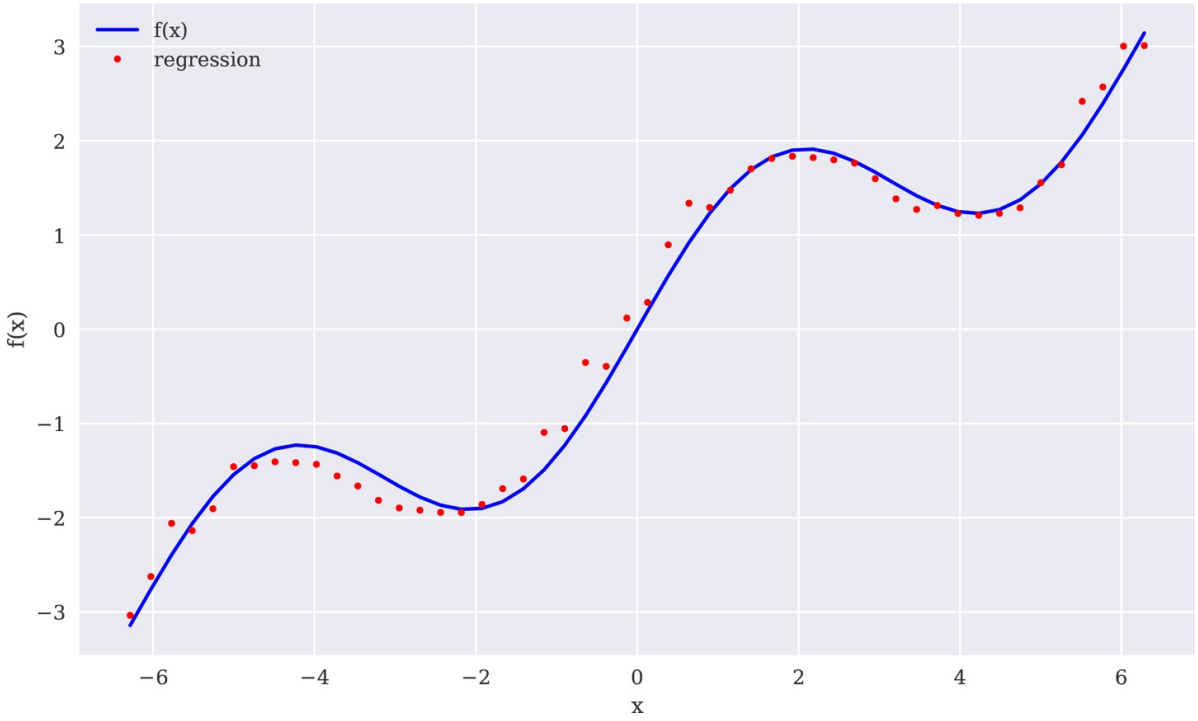

Figure 11-7. Regression for noisy data

Unsorted data

Another important aspect of regression is that the approach also works seamlessly with unsorted data. The previous examples all rely on sorted *x* data. This does not have to be the case. To make the point, let's look at yet another randomization approach for the *x* values. In this case, one can hardly identify any structure by just visually inspecting the raw data:

```
In [32]: xu = np.random.rand(50) * 4 * np.pi - 2 * np.pi  ❶
```

```
          yu = f(xu)

In [33]: print(xu[:10].round(2))  ❶
          print(yu[:10].round(2))  ❶
          [-4.17 -0.11 -1.91  2.33  3.34 -0.96  5.81  4.92 -4.56 -5.42]
          [-1.23 -0.17 -1.9   1.89  1.47 -1.29  2.45  1.48 -1.29 -1.95]

In [34]: reg = np.polyfit(xu, yu, 5)
          ry = np.polyval(reg, xu)

In [35]: create_plot([xu, xu], [yu, ry], ['b.', 'ro'],
                     ['f(x)', 'regression'], ['x', 'f(x)'])
```

❶ Randomizes the *x* values.

As with the noisy data, the regression approach does not care for the order of the observation points. This becomes obvious upon inspecting the structure of the minimization problem in Equation 11-1. It is also obvious by the results, presented in Figure 11-8.

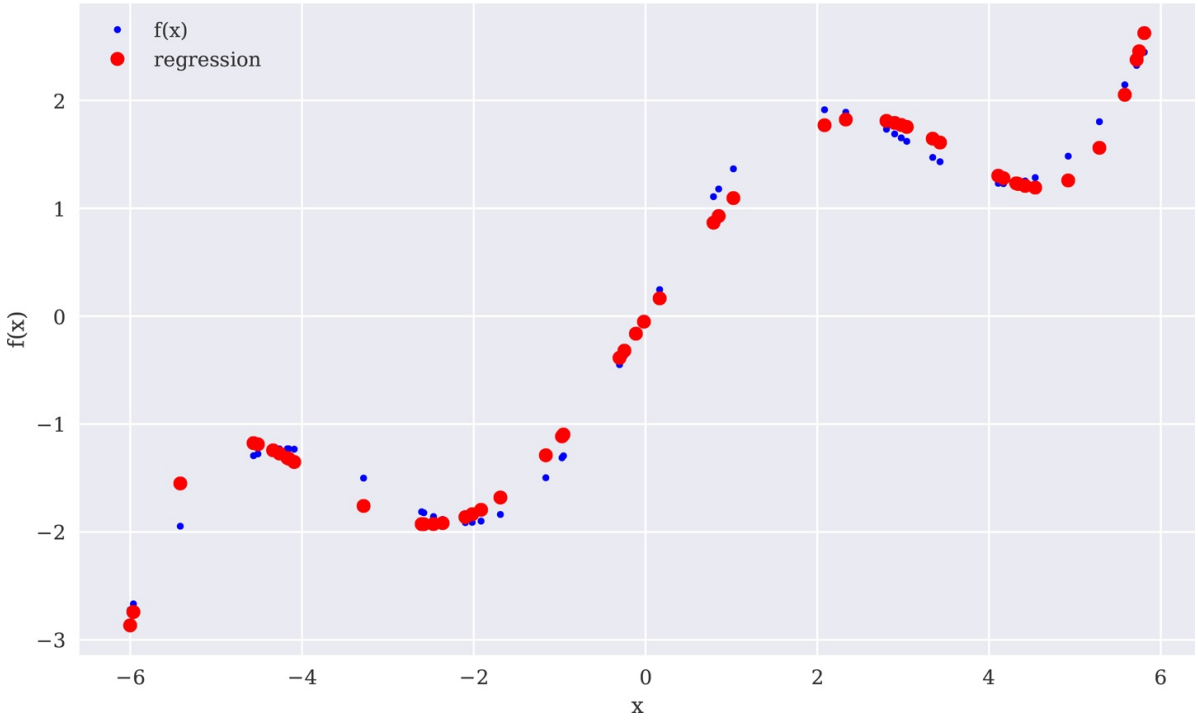

Figure 11-8. Regression for unsorted data

Multiple dimensions

Another convenient characteristic of the least-squares regression approach is that it carries over to multiple dimensions without too many modifications. As an

example function take `fm()`, as presented next:

```
In [36]: def fm(p):
             x, y = p
             return np.sin(x) + 0.25 * x + np.sqrt(y) + 0.05 * y ** 2
```

To properly visualize this function, *grids* (in two dimensions) of independent data points are needed. Based on such two-dimensional grids of independent and resulting dependent data points, embodied in the following by `X`, `Y`, and `Z`, Figure 11-9 presents the shape of the function `fm()`:

```
In [37]: x = np.linspace(0, 10, 20)
         y = np.linspace(0, 10, 20)
         X, Y = np.meshgrid(x, y)  ❶

In [38]: Z = fm((X, Y))
         x = X.flatten()  ❷
         y = Y.flatten()  ❷

In [39]: from mpl_toolkits.mplot3d import Axes3D  ❸

In [40]: fig = plt.figure(figsize=(10, 6))
         ax = fig.gca(projection='3d')
         surf = ax.plot_surface(X, Y, Z, rstride=2, cstride=2,
                                cmap='coolwarm', linewidth=0.5,
                                antialiased=True)
         ax.set_xlabel('x')
         ax.set_ylabel('y')
         ax.set_zlabel('f(x, y)')
         fig.colorbar(surf, shrink=0.5, aspect=5)
```

❶ Generates 2D `ndarray` objects ("grids") out of the 1D `ndarray` objects.

❷ Yields 1D `ndarray` objects from the 2D `ndarray` objects.

❸ Imports the 3D plotting capabilities from `matplotlib` as required.

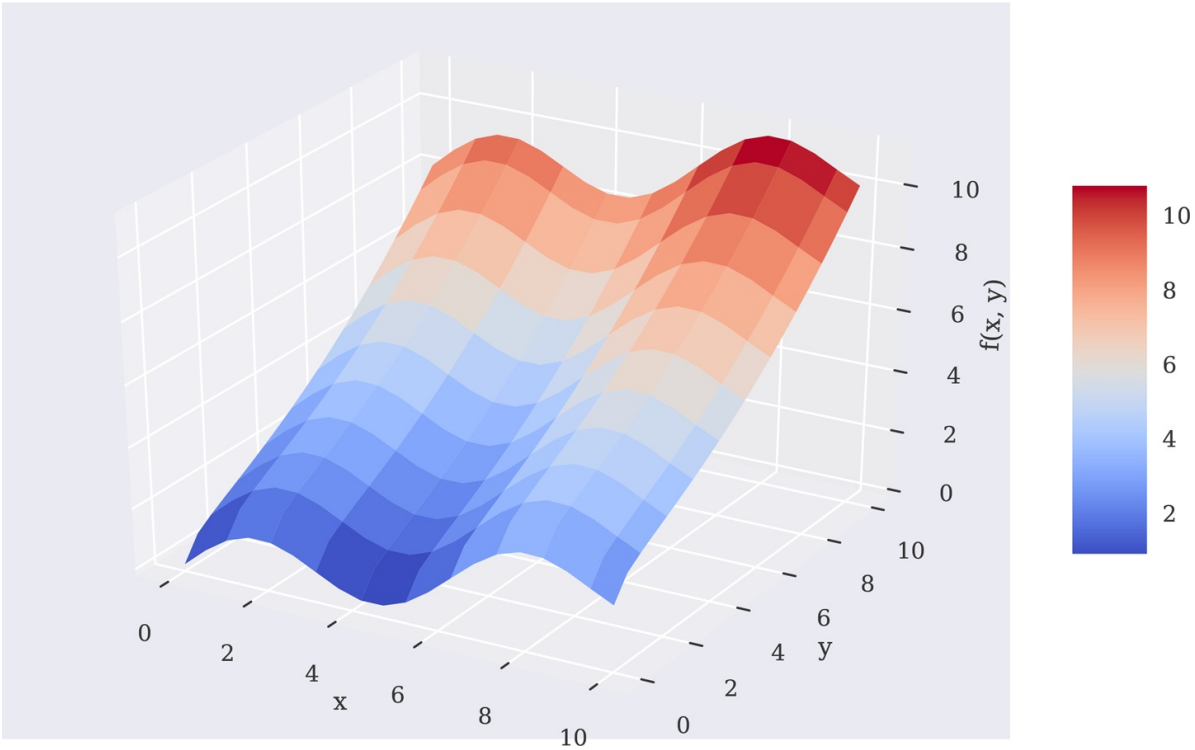

Figure 11-9. The function with two parameters

To get good regression results, the set of basis functions is essential. Therefore, factoring in knowledge about the function `fm()` itself, both an `np.sin()` and an `np.sqrt()` function are included. Figure 11-10 shows the perfect regression results visually:

```
In [41]: matrix = np.zeros((len(x), 6 + 1))
         matrix[:, 6] = np.sqrt(y)  ❶
         matrix[:, 5] = np.sin(x)  ❷
         matrix[:, 4] = y ** 2
         matrix[:, 3] = x ** 2
         matrix[:, 2] = y
         matrix[:, 1] = x
         matrix[:, 0] = 1

In [42]: reg = np.linalg.lstsq(matrix, fm((x, y)), rcond=None)[0]

In [43]: RZ = np.dot(matrix, reg).reshape((20, 20))  ❸

In [44]: fig = plt.figure(figsize=(10, 6))
         ax = fig.gca(projection='3d')
         surf1 = ax.plot_surface(X, Y, Z, rstride=2, cstride=2,
                     cmap=mpl.cm.coolwarm, linewidth=0.5,
                     antialiased=True)  ❹
         surf2 = ax.plot_wireframe(X, Y, RZ, rstride=2, cstride=2,
```

```
                                label='regression')  ❺
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('f(x, y)')
ax.legend()
fig.colorbar(surf, shrink=0.5, aspect=5)
```

❶ The `np.sqrt()` function for the `y` parameter.

❷ The `np.sin()` function for the `x` parameter.

❸ Transforms the regression results to the grid structure.

❹ Plots the original function surface.

❺ Plots the regression surface.

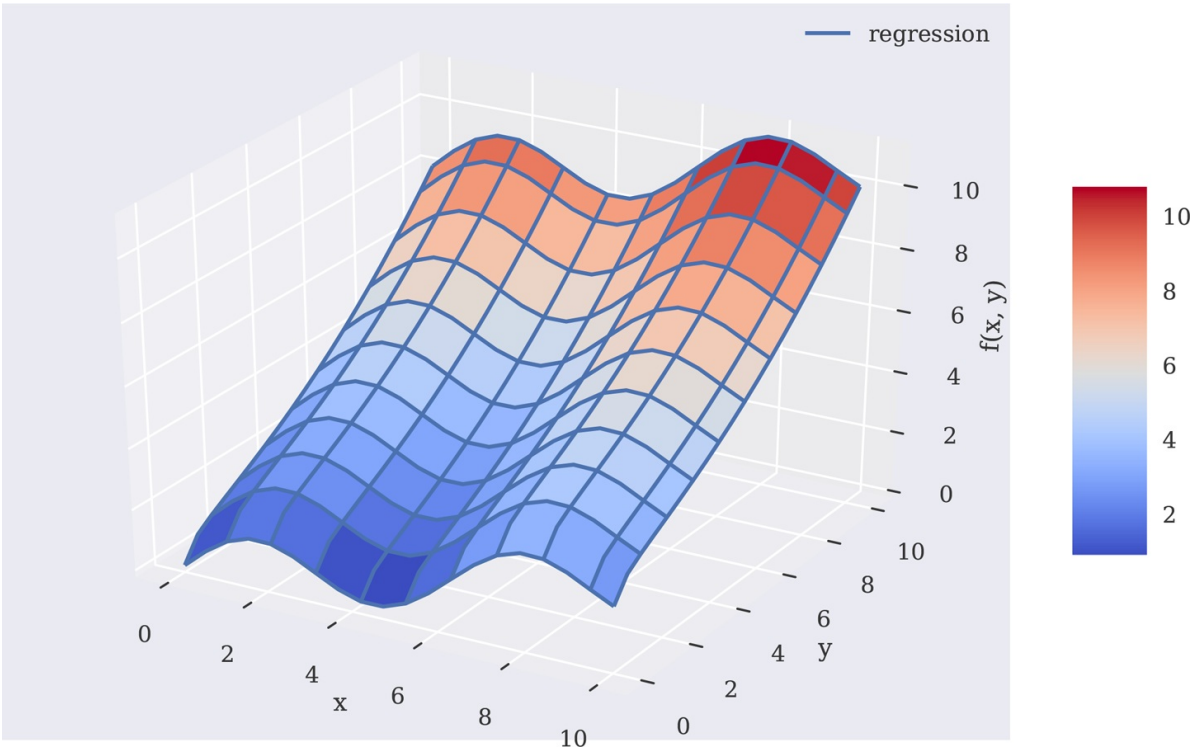

Figure 11-10. Regression surface for function with two parameters

REGRESSION

Least-squares regression approaches have multiple areas of application, including simple function approximation and function approximation based on noisy or unsorted data. These approaches can be applied to one-dimensional as well as multidimensional problems. Due to the underlying mathematics, the application is “almost always the same.”

Interpolation

Compared to regression, *interpolation* (e.g., with cubic splines) is more involved mathematically. It is also limited to low-dimensional problems. Given an ordered set of observation points (ordered in the *x* dimension), the basic idea is to do a regression between two neighboring data points in such a way that not only are the data points perfectly matched by the resulting piecewise-defined interpolation function, but also the function is continuously differentiable at the data points. Continuous differentiability requires at least interpolation of degree 3—i.e., with cubic splines. However, the approach also works in general with quadratic and even linear splines.

The following code implements a linear splines interpolation, the result of which is shown in Figure 11-11:

```
In [45]: import scipy.interpolate as spi  ❶

In [46]: x = np.linspace(-2 * np.pi, 2 * np.pi, 25)

In [47]: def f(x):
             return np.sin(x) + 0.5 * x

In [48]: ipo = spi.splrep(x, f(x), k=1)  ❷

In [49]: iy = spi.splev(x, ipo)  ❸

In [50]: np.allclose(f(x), iy)  ❹
Out[50]: True

In [51]: create_plot([x, x], [f(x), iy], ['b', 'ro'],
                     ['f(x)', 'interpolation'], ['x', 'f(x)'])
```

❶ Imports the required subpackage from `SciPy`.
❷ Implements a linear spline interpolation.
❸ Derives the interpolated values.
❹ Checks whether the interpolated values are close (enough) to the function values.

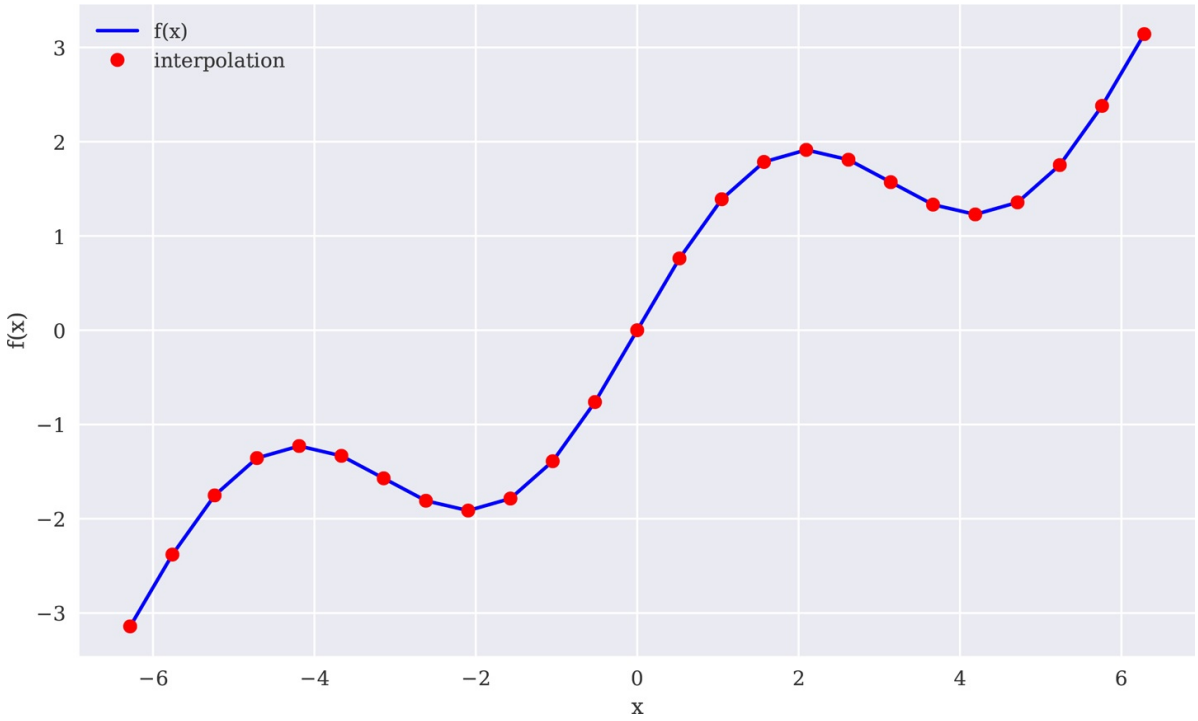

Figure 11-11. Linear splines interpolation (complete data set)

The application itself, given an *x*-ordered set of data points, is as simple as the application of `np.polyfit()` and `np.polyval()`. Here, the respective functions are `sci.splrep()` and `sci.splev()`. Table 11-2 lists the major parameters that the `sci.splrep()` function takes.

Table 11-2. Parameters of splrep() function

Parameter	Description
`x`	(Ordered) *x* coordinates (independent variable values)
`y`	(*x*-ordered) *y* coordinates (dependent variable values)
`w`	Weights to apply to the *y* coordinates
`xb`, `xe`	Interval to fit; if `None` then `[x[0], x[-1]]`
`k`	Order of the spline fit ($1 \leq k \leq 5$)
`s`	Smoothing factor (the larger, the more smoothing)
`full_output`	If `True`, returns additional output
`quiet`	If `True`, suppresses messages

Table 11-3 lists the parameters that the `sci.splev()` function takes.

Table 11-3. Parameters of splev() function

Parameter	Description
`x`	(Ordered) *x* coordinates (independent variable values)
`tck`	Sequence of length 3 returned by `splrep()` (knots, coefficients, degree)
`der`	Order of derivative (`0` for function, `1` for first derivative)
`ext`	Behavior if `x` not in knot sequence (`0` = extrapolate, `1` = return 0, `2` = raise `ValueError`)

Spline interpolation is often used in finance to generate estimates for dependent values of independent data points not included in the original observations. To this end, the next example picks a much smaller interval and has a closer look at the interpolated values with the linear splines. Figure 11-12 reveals that the interpolation function indeed interpolates *linearly* between two observation points. For certain applications this might not be precise enough. In addition, it is evident that the function is not continuously differentiable at the original data points—another drawback:

```
In [52]: xd = np.linspace(1.0, 3.0, 50)  ❶
         iyd = spi.splev(xd, ipo)

In [53]: create_plot([xd, xd], [f(xd), iyd], ['b', 'ro'],
                     ['f(x)', 'interpolation'], ['x', 'f(x)'])
```

❶ Smaller interval with more points.

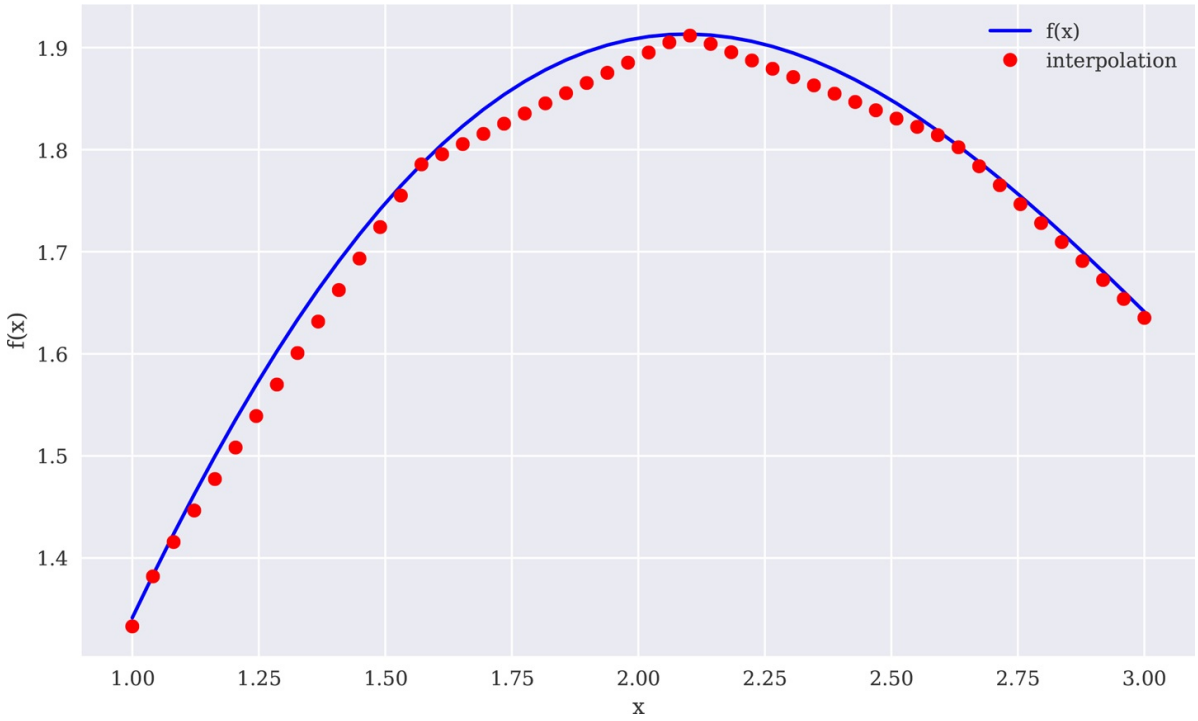

Figure 11-12. Linear splines interpolation (data subset)

A repetition of the complete exercise, this time using cubic splines, improves the results considerably (see Figure 11-13):

```
In [54]: ipo = spi.splrep(x, f(x), k=3)  ❶
         iyd = spi.splev(xd, ipo)  ❷

In [55]: np.allclose(f(xd), iyd)  ❸
Out[55]: False

In [56]: np.mean((f(xd) - iyd) ** 2)  ❹
Out[56]: 1.1349319851436892e-08

In [57]: create_plot([xd, xd], [f(xd), iyd], ['b', 'ro'],
                     ['f(x)', 'interpolation'], ['x', 'f(x)'])
```

❶ Cubic splines interpolation on complete data sets.

❷ Results applied to the smaller interval.

❸ The interpolation is still not perfect …

❹ … but better than before.

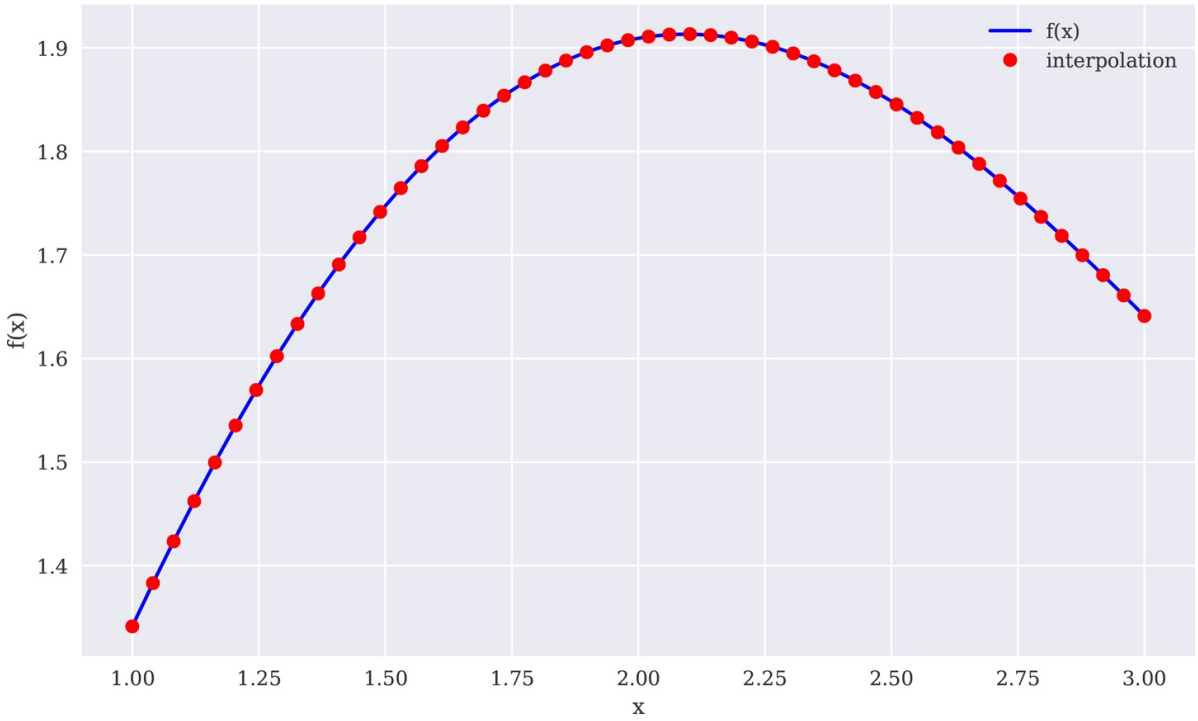

Figure 11-13. Cubic splines interpolation (data subset)

> **INTERPOLATION**
>
> In those cases where spline interpolation can be applied, one can expect better approximation results compared to a least-squares regression approach. However, remember that sorted (and "non-noisy") data is required and that the approach is limited to low-dimensional problems. It is also computationally more demanding and might therefore take (much) longer than regression in certain use cases.

Convex Optimization

In finance and economics, *convex optimization* plays an important role. Examples are the calibration of option pricing models to market data or the optimization of an agent's utility function. As an example, take the function `fm()`:

```
In [58]: def fm(p):
             x, y = p
             return (np.sin(x) + 0.05 * x ** 2
                     + np.sin(y) + 0.05 * y ** 2)
```

Figure 11-14 shows the function graphically for the defined intervals for `x` and `y`. Visual inspection already reveals that this function has multiple local minima. The existence of a global minimum cannot really be confirmed by this particular graphical representation, but it seems to exist:

```
In [59]: x = np.linspace(-10, 10, 50)
         y = np.linspace(-10, 10, 50)
         X, Y = np.meshgrid(x, y)
         Z = fm((X, Y))

In [60]: fig = plt.figure(figsize=(10, 6))
         ax = fig.gca(projection='3d')
         surf = ax.plot_surface(X, Y, Z, rstride=2, cstride=2,
                                cmap='coolwarm', linewidth=0.5,
                                antialiased=True)
         ax.set_xlabel('x')
         ax.set_ylabel('y')
         ax.set_zlabel('f(x, y)')
         fig.colorbar(surf, shrink=0.5, aspect=5)
```

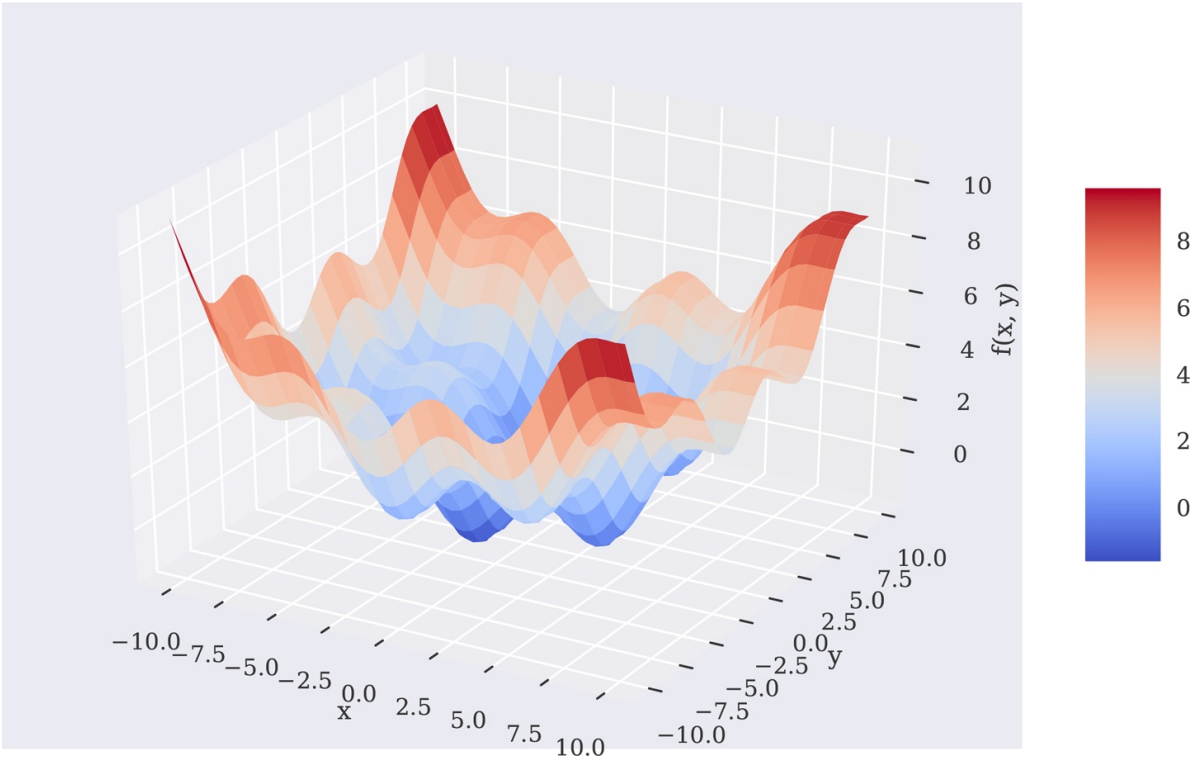

Figure 11-14. Linear splines interpolation (data subset)

Global Optimization

In what follows, both a *global* minimization approach and a *local* one are implemented. The functions `sco.brute()` and `sco.fmin()` that are applied are from `scipy.optimize`.

To have a closer look behind the scenes during minimization procedures, the following code amends the original function by an option to output current parameter values as well as the function value. This allows us to keep track of all relevant information for the procedure:

```
In [61]: import scipy.optimize as sco  ❶

In [62]: def fo(p):
             x, y = p
             z = np.sin(x) + 0.05 * x ** 2 + np.sin(y) + 0.05 * y ** 2
             if output == True:
                 print('%8.4f | %8.4f | %8.4f' % (x, y, z))  ❷
             return z

In [63]: output = True
         sco.brute(fo, ((-10, 10.1, 5), (-10, 10.1, 5)), finish=None)  ❸
         -10.0000 | -10.0000 |  11.0880
         -10.0000 | -10.0000 |  11.0880
         -10.0000 |  -5.0000 |   7.7529
         -10.0000 |   0.0000 |   5.5440
         -10.0000 |   5.0000 |   5.8351
         -10.0000 |  10.0000 |  10.0000
          -5.0000 | -10.0000 |   7.7529
          -5.0000 |  -5.0000 |   4.4178
          -5.0000 |   0.0000 |   2.2089
          -5.0000 |   5.0000 |   2.5000
          -5.0000 |  10.0000 |   6.6649
           0.0000 | -10.0000 |   5.5440
           0.0000 |  -5.0000 |   2.2089
           0.0000 |   0.0000 |   0.0000
           0.0000 |   5.0000 |   0.2911
           0.0000 |  10.0000 |   4.4560
           5.0000 | -10.0000 |   5.8351
           5.0000 |  -5.0000 |   2.5000
           5.0000 |   0.0000 |   0.2911
           5.0000 |   5.0000 |   0.5822
           5.0000 |  10.0000 |   4.7471
          10.0000 | -10.0000 |  10.0000
          10.0000 |  -5.0000 |   6.6649
          10.0000 |   0.0000 |   4.4560
          10.0000 |   5.0000 |   4.7471
          10.0000 |  10.0000 |   8.9120
```

```
Out[63]: array([0., 0.])
```

❶ Imports the required subpackage from `SciPy`.

❷ The information to print out if `output = True`.

❸ The brute force optimization.

The optimal parameter values, given the initial parameterization of the function, are `x = y = 0`. The resulting function value is also `0`, as a quick review of the preceding output reveals. One might be inclined to accept this as the global minimum. However, the first parameterization here is quite rough, in that step sizes of 5 for both input parameters are used. This can of course be refined considerably, leading to better results in this case—and showing that the previous solution is not the optimal one:

```
In [64]: output = False
         opt1 = sco.brute(fo, ((-10, 10.1, 0.1), (-10, 10.1, 0.1)), finish=None)

In [65]: opt1
Out[65]: array([-1.4, -1.4])

In [66]: fm(opt1)
Out[66]: -1.7748994599769203
```

The optimal parameter values are now `x = y = -1.4` and the minimal function value for the global minimization is about –1.7749.

Local Optimization

The local convex optimization that follows draws on the results from the global optimization. The function `sco.fmin()` takes as input the function to minimize and the starting parameter values. Optional parameter values are the input parameter tolerance and function value tolerance, as well as the maximum number of iterations and function calls. The local optimization further improves the result:

```
In [67]: output = True
         opt2 = sco.fmin(fo, opt1, xtol=0.001, ftol=0.001,
                         maxiter=15, maxfun=20)  ❶
          -1.4000 |  -1.4000 |  -1.7749
          -1.4700 |  -1.4000 |  -1.7743
```

```
 -1.4000 |  -1.4700 |  -1.7743
 -1.3300 |  -1.4700 |  -1.7696
 -1.4350 |  -1.4175 |  -1.7756
 -1.4350 |  -1.3475 |  -1.7722
 -1.4088 |  -1.4394 |  -1.7755
 -1.4438 |  -1.4569 |  -1.7751
 -1.4328 |  -1.4427 |  -1.7756
 -1.4591 |  -1.4208 |  -1.7752
 -1.4213 |  -1.4347 |  -1.7757
 -1.4235 |  -1.4096 |  -1.7755
 -1.4305 |  -1.4344 |  -1.7757
 -1.4168 |  -1.4516 |  -1.7753
 -1.4305 |  -1.4260 |  -1.7757
 -1.4396 |  -1.4257 |  -1.7756
 -1.4259 |  -1.4325 |  -1.7757
 -1.4259 |  -1.4241 |  -1.7757
 -1.4304 |  -1.4177 |  -1.7757
 -1.4270 |  -1.4288 |  -1.7757
Warning: Maximum number of function evaluations has been exceeded.
```

```
In [68]: opt2
Out[68]: array([-1.42702972, -1.42876755])

In [69]: fm(opt2)
Out[69]: -1.7757246992239009
```

❶ The local convex optimization.

For many convex optimization problems it is advisable to have a global minimization before the local one. The major reason for this is that local convex optimization algorithms can easily be trapped in a local minimum (or do “basin hopping”), ignoring completely better local minima and/or a global minimum. The following shows that setting the starting parameterization to `x = y = 2` gives, for example, a “minimum” value of above zero:

```
In [70]: output = False
         sco.fmin(fo, (2.0, 2.0), maxiter=250)
         Optimization terminated successfully.
                  Current function value: 0.015826
                  Iterations: 46
                  Function evaluations: 86

Out[70]: array([4.2710728 , 4.27106945])
```

Constrained Optimization

So far, this section only considers unconstrained optimization problems. However, large classes of economic or financial optimization problems are constrained by one or multiple constraints. Such constraints can formally take on the form of equalities or inequalities.

As a simple example, consider the utility maximization problem of an (expected utility maximizing) investor who can invest in two risky securities. Both securities cost $q_a = q_b = 10$ USD today. After one year, they have a payoff of 15 USD and 5 USD, respectively, in state *u*, and of 5 USD and 12 USD, respectively, in state *d*. Both states are equally likely. Denote the vector payoffs for the two securities by r_a and r_b, respectively.

The investor has a budget of $w_0 = 100$ USD to invest and derives utility from future wealth according to the utility function $u(w) = \sqrt{w}$, where *w* is the wealth (USD amount) available. Equation 11-2 is a formulation of the maximization problem where *a*, *b* are the numbers of securities bought by the investor.

Equation 11-2. Expected utility maximization problem (1)

$$\begin{aligned} \max_{a,b} \mathbf{E}\left(u\left(w_1\right)\right) &= p\sqrt{w_{1u}} + (1-p)\sqrt{w_{1d}} \\ w_1 &= a \cdot r_a + b \cdot r_b \\ w_0 &\geq a \cdot q_a + b \cdot q_b \\ a, b &\geq 0 \end{aligned}$$

Putting in all numerical assumptions, one gets the problem in Equation 11-3. Note the change to the minimization of the negative expected utility.

Equation 11-3. Expected utility maximization problem (2)

$$
\begin{aligned}
\min_{a,b} -\mathbf{E}\left(u\left(w_1\right)\right) &= -\left(0.5 \cdot \sqrt{w_{1u}} + 0.5 \cdot \sqrt{w_{1d}}\right) \\
w_{1u} &= a \cdot 15 + b \cdot 5 \\
w_{1d} &= a \cdot 5 + b \cdot 12 \\
100 &\geq a \cdot 10 + b \cdot 10 \\
a, b &\geq 0
\end{aligned}
$$

To solve this problem, the `scipy.optimize.minimize()` function is appropriate. This function takes as input—in addition to the function to be minimized—conditions in the form of equalities and inequalities (as a `list` of `dict` objects) as well as boundaries for the parameters (as a `tuple` of `tuple` objects).[1] The following translates the problem from Equation 11-3 into Python code:

```
In [71]: import math

In [72]: def Eu(p):  ❶
             s, b = p
             return -(0.5 * math.sqrt(s * 15 + b * 5) +
                      0.5 * math.sqrt(s * 5 + b * 12))

In [73]: cons = ({'type': 'ineq',
                  'fun': lambda p: 100 - p[0] * 10 - p[1] * 10})  ❷

In [74]: bnds = ((0, 1000), (0, 1000))  ❸

In [75]: result = sco.minimize(Eu, [5, 5], method='SLSQP',
                               bounds=bnds, constraints=cons)  ❹
```

❶ The function to be *minimized*, in order to maximize the expected utility.

❷ The inequality constraint as a `dict` object.

❸ The boundary values for the parameters (chosen to be wide enough).

❹ The constrained optimization.

The `result` object contains all the relevant information. With regard to the minimal function value, one needs to recall to shift the sign back:

```
In [76]: result
Out[76]:      fun: -9.700883611487832
```

```
             jac: array([-0.48508096, -0.48489535])
         message: 'Optimization terminated successfully.'
            nfev: 21
             nit: 5
            njev: 5
          status: 0
         success: True
               x: array([8.02547122, 1.97452878])

In [77]: result['x']  ❶
Out[77]: array([8.02547122, 1.97452878])

In [78]: -result['fun']  ❷
Out[78]: 9.700883611487832

In [79]: np.dot(result['x'], [10, 10])  ❸
Out[79]: 99.99999999999999
```

❶ The optimal parameter values (i.e., the optimal portfolio).

❷ The negative minimum function value as the optimal solution value.

❸ The budget constraint is binding; all wealth is invested.

Integration

Especially when it comes to valuation and option pricing, integration is an important mathematical tool. This stems from the fact that risk-neutral values of derivatives can be expressed in general as the discounted *expectation* of their payoff under the risk-neutral or martingale measure. The expectation in turn is a sum in the discrete case and an integral in the continuous case. The subpackage `scipy.integrate` provides different functions for numerical integration. The example function is known from “Approximation”:

```
In [80]: import scipy.integrate as sci

In [81]: def f(x):
             return np.sin(x) + 0.5 * x
```

The integration interval shall be `[0.5, 9.5]`, leading to the definite integral as in Equation 11-4.

Equation 11-4. Integral of example function

$$\int_{0.5}^{9.5} f(x)\,dx = \int_{0.5}^{9.5} \sin(x) + \frac{x}{2}\,dx$$

The following code defines the major Python objects to evaluate the integral:

```
In [82]: x = np.linspace(0, 10)
         y = f(x)
         a = 0.5  ❶
         b = 9.5  ❷
         Ix = np.linspace(a, b)  ❸
         Iy = f(Ix)  ❹
```

❶ Left integration limit.
❷ Right integration limit.
❸ Integration interval values.
❹ Integration function values.

Figure 11-15 visualizes the integral value as the gray-shaded area under the function:[2]

```
In [83]: from matplotlib.patches import Polygon

In [84]: fig, ax = plt.subplots(figsize=(10, 6))
         plt.plot(x, y, 'b', linewidth=2)
         plt.ylim(bottom=0)
         Ix = np.linspace(a, b)
         Iy = f(Ix)
         verts = [(a, 0)] + list(zip(Ix, Iy)) + [(b, 0)]
         poly = Polygon(verts, facecolor='0.7', edgecolor='0.5')
         ax.add_patch(poly)
         plt.text(0.75 * (a + b), 1.5, r"$\int_a^b f(x)dx$",
                  horizontalalignment='center', fontsize=20)
         plt.figtext(0.9, 0.075, '$x$')
         plt.figtext(0.075, 0.9, '$f(x)$')
         ax.set_xticks((a, b))
         ax.set_xticklabels(('$a$', '$b$'))
         ax.set_yticks([f(a), f(b)]);
```

Figure 11-15. Integral value as shaded area

Numerical Integration

The `scipy.integrate` subpackage contains a selection of functions to numerically integrate a given mathematical function for upper and lower integration limits. Examples are `sci.fixed_quad()` for *fixed Gaussian quadrature*, `sci.quad()` for *adaptive quadrature*, and `sci.romberg()` for *Romberg integration*:

```
In [85]: sci.fixed_quad(f, a, b)[0]
Out[85]: 24.366995967084602

In [86]: sci.quad(f, a, b)[0]
Out[86]: 24.374754718086752

In [87]: sci.romberg(f, a, b)
Out[87]: 24.374754718086713
```

There are also a number of integration functions that take as input `list` or `ndarray` objects with function values and input values, respectively. Examples in this regard are `sci.trapz()`, using the *trapezoidal* rule, and `sci.simps()`, implementing *Simpson's* rule:

```
In [88]: xi = np.linspace(0.5, 9.5, 25)

In [89]: sci.trapz(f(xi), xi)
Out[89]: 24.352733271544516

In [90]: sci.simps(f(xi), xi)
Out[90]: 24.37496418455075
```

Integration by Simulation

The valuation of options and derivatives by Monte Carlo simulation (see Chapter 12) rests on the insight that one can evaluate an integral by simulation. To this end, draw *I* random values of `x` between the integral limits and evaluate the integration function at every random value for `x`. Sum up all the function values and take the average to arrive at an average function value over the integration interval. Multiply this value by the length of the integration interval to derive an estimate for the integral value.

The following code shows how the Monte Carlo estimated integral value converges—although not monotonically—to the real one when one increases the number of random draws. The estimator is already quite close for relatively small numbers of random draws:

```
In [91]: for i in range(1, 20):
             np.random.seed(1000)
             x = np.random.random(i * 10) * (b - a) + a  ❶
             print(np.mean(f(x)) * (b - a))
         24.804762279331463
         26.522918898332378
         26.265547519223976
         26.02770339943824
         24.99954181440844
         23.881810141621663
         23.527912274843253
         23.507857658961207
         23.67236746066989
         23.679410416062886
         24.424401707879305
         24.239005346819056
         24.115396924962802
         24.424191987566726
         23.924933080533783
         24.19484212027875
```

```
24.117348378249833
24.100690929662274
23.76905109847816
```

❶ Number of random `x` values is increased with every iteration.

Symbolic Computation

The previous sections are mainly concerned with numerical computation. This section now introduces *symbolic* computation, which can be applied beneficially in many areas of finance. To this end, `SymPy`, a library specifically dedicated to symbolic computation, is generally used.

Basics

`SymPy` introduces new classes of objects. A fundamental class is the `Symbol` class:

```
In [92]: import sympy as sy

In [93]: x = sy.Symbol('x')  ❶
         y = sy.Symbol('y')  ❶

In [94]: type(x)
Out[94]: sympy.core.symbol.Symbol

In [95]: sy.sqrt(x)  ❷
Out[95]: sqrt(x)

In [96]: 3 + sy.sqrt(x) - 4 ** 2  ❸
Out[96]: sqrt(x) - 13

In [97]: f = x ** 2 + 3 + 0.5 * x ** 2 + 3 / 2  ❹

In [98]: sy.simplify(f)  ❺
Out[98]: 1.5*x**2 + 4.5
```

❶ Defines symbols to work with.
❷ Applies a function on a symbol.
❸ A numerical expression defined on symbol.
❹ A function defined symbolically.

❺ The function expression simplified.

This already illustrates a major difference to regular Python code. Although `x` has no numerical value, the square root of `x` is nevertheless defined with `SymPy` since `x` is a `Symbol` object. In that sense, `sy.sqrt(x)` can be part of arbitrary mathematical expressions. Notice that `SymPy` in general automatically simplifies a given mathematical expression. Similarly, one can define arbitrary functions using `Symbol` objects. They are not to be confused with Python functions.

`SymPy` provides three basic renderers for mathematical expressions:

- LaTeX-based
- Unicode-based
- ASCII-based

When working, for example, solely in a Jupyter Notebook environment (HTML-based), LaTeX rendering is generally a good (i.e., visually appealing) choice. The code that follows sticks to the simplest option, ASCII, to illustrate that there is no manual typesetting involved:

```
In [99]: sy.init_printing(pretty_print=False, use_unicode=False)

In [100]: print(sy.pretty(f))
                2
          1.5*x  + 4.5

In [101]: print(sy.pretty(sy.sqrt(x) + 0.5))
          ___
          \/ x  + 0.5
```

This section cannot go into details, but `SymPy` also provides many other useful mathematical functions—for example, when it comes to numerically evaluating π. The following example shows the first and final 40 characters of the string representation of π up to the 400,000th digit. It also searches for a six-digit, day-first birthday—a popular task in certain mathematics and IT circles:

```
In [102]: %time pi_str = str(sy.N(sy.pi, 400000))  ❶
          CPU times: user 400 ms, sys: 10.9 ms, total: 411 ms
          Wall time: 501 ms
```

```
In [103]: pi_str[:42]  ❷
Out[103]: '3.14159265358979323846264338327950288419'

In [104]: pi_str[-40:]  ❸
Out[104]: '8245672736856312185020980470362464176198'

In [105]: %time pi_str.find('061072')  ❹
          CPU times: user 115 µs, sys: 1e+03 ns, total: 116 µs
          Wall time: 120 µs

Out[105]: 80847
```

❶ Returns the string representation of the first 400,000 digits of π.
❷ Shows the first 40 digits …
❸ … and the final 40 digits.
❹ Searches for a birthday date in the string.

Equations

A strength of SymPy is solving equations, e.g., of the form $x^2 - 1 = 0$. In general, SymPy presumes that one is looking for a solution to the equation obtained by equating the given expression to zero. Therefore, equations like $x^2 - 1 = 3$ might have to be reformulated to get the desired result. Of course, SymPy can cope with more complex expressions, like $x^3 + 0.5x^2 - 1 = 0$. Finally, it can also deal with problems involving imaginary numbers, such as $x^2 + y^2 = 0$:

```
In [106]: sy.solve(x ** 2 - 1)
Out[106]: [-1, 1]

In [107]: sy.solve(x ** 2 - 1 - 3)
Out[107]: [-2, 2]

In [108]: sy.solve(x ** 3 + 0.5 * x ** 2 - 1)
Out[108]: [0.858094329496553, -0.679047164748276 - 0.839206763026694*I,
           -0.679047164748276 + 0.839206763026694*I]

In [109]: sy.solve(x ** 2 + y ** 2)
Out[109]: [{x: -I*y}, {x: I*y}]
```

Integration and Differentiation

Another strength of `SymPy` is integration and differentiation. The example that follows revisits the example function used for numerical- and simulation-based integration and derives both a *symbolically* and a *numerically* exact solution. `Symbol` objects for the integration limits objects are required to get started:

```
In [110]: a, b = sy.symbols('a b')  ❶

In [111]: I = sy.Integral(sy.sin(x) + 0.5 * x, (x, a, b))  ❷

In [112]: print(sy.pretty(I))  ❷
            b
            /
           |
           |  (0.5*x + sin(x)) dx
           |
          /
          a

In [113]: int_func = sy.integrate(sy.sin(x) + 0.5 * x, x)  ❸

In [114]: print(sy.pretty(int_func))  ❸
                2
          0.25*x  - cos(x)

In [115]: Fb = int_func.subs(x, 9.5).evalf()  ❹
          Fa = int_func.subs(x, 0.5).evalf()  ❹

In [116]: Fb - Fa  ❺
Out[116]: 24.3747547180867
```

❶ The `Symbol` objects for the integral limits.

❷ The `Integral` object defined and pretty-printed.

❸ The antiderivative derived and pretty-printed.

❹ The values of the antiderivative at the limits, obtained via the `.subs()` and `.evalf()` methods.

❺ The exact numerical value of the integral.

The integral can also be solved symbolically with the symbolic integration limits:

```
In [117]: int_func_limits = sy.integrate(sy.sin(x) + 0.5 * x, (x, a, b))  ❶

In [118]: print(sy.pretty(int_func_limits))  ❶
```

```
                2         2
          - 0.25*a  + 0.25*b  + cos(a) - cos(b)

In [119]: int_func_limits.subs({a : 0.5, b : 9.5}).evalf()  ❷
Out[119]: 24.3747547180868

In [120]: sy.integrate(sy.sin(x) + 0.5 * x, (x, 0.5, 9.5))  ❸
Out[120]: 24.3747547180867
```

❶ Solving the integral symbolically.

❷ Solving the integral numerically, using a `dict` object during substitution.

❸ Solving the integral numerically in a single step.

Differentiation

The derivative of the antiderivative yields in general the original function. Applying the `sy.diff()` function to the symbolic antiderivative illustrates this:

```
In [121]: int_func.diff()
Out[121]: 0.5*x + sin(x)
```

As with the integration example, differentiation shall now be used to derive the exact solution of the convex minimization problem this chapter looked at earlier. To this end, the respective function is defined symbolically, partial derivatives are derived, and the roots are identified.

A necessary but not sufficient condition for a global minimum is that both partial derivatives are zero. However, there is no guarantee of a symbolic solution. Both algorithmic and (multiple) existence issues come into play here. However, one can solve the two first-order conditions numerically, providing “educated” guesses based on the global and local minimization efforts from before:

```
In [122]: f = (sy.sin(x) + 0.05 * x ** 2
             + sy.sin(y) + 0.05 * y ** 2)  ❶

In [123]: del_x = sy.diff(f, x)  ❷
          del_x  ❷
Out[123]: 0.1*x + cos(x)

In [124]: del_y = sy.diff(f, y)  ❷
          del_y  ❷
Out[124]: 0.1*y + cos(y)
```

```
In [125]: xo = sy.nsolve(del_x, -1.5)  ❸
          xo  ❸
Out[125]: -1.42755177876459

In [126]: yo = sy.nsolve(del_y, -1.5)  ❸
          yo  ❸
Out[126]: -1.42755177876459

In [127]: f.subs({x : xo, y : yo}).evalf()  ❹
Out[127]: -1.77572565314742
```

❶ The symbolic version of the function.
❷ The two partial derivatives derived and printed.
❸ Educated guesses for the roots and resulting optimal values.
❹ The global minimum function value.

Again, providing uneducated/arbitrary guesses might trap the algorithm in a local minimum instead of the global one:

```
In [128]: xo = sy.nsolve(del_x, 1.5)  ❶
          xo
Out[128]: 1.74632928225285

In [129]: yo = sy.nsolve(del_y, 1.5)  ❶
          yo
Out[129]: 1.74632928225285

In [130]: f.subs({x : xo, y : yo}).evalf()  ❷
Out[130]: 2.27423381055640
```

❶ Uneducated guesses for the roots.
❷ The local minimum function value.

This numerically illustrates that the first-order conditions are necessary but not sufficient.

SYMBOLIC COMPUTATIONS

When doing (financial) mathematics with Python, `SymPy` and symbolic computations prove to be a valuable tool. Especially for interactive financial analytics, this can be a more efficient approach compared to nonsymbolic approaches.

Conclusion

This chapter covers selected mathematical topics and tools important to finance. For example, the approximation of functions is important in many financial areas, like factor-based models, yield curve interpolation, and regression-based Monte Carlo valuation approaches for American options. Convex optimization techniques are also regularly needed in finance; for example, when calibrating parametric option pricing models to market quotes or implied volatilities of options.

Numerical integration is central to, for example, the pricing of options and derivatives. Having derived the risk-neutral probability measure for a (set of) stochastic process(es), option pricing boils down to taking the expectation of the option's payoff under the risk-neutral measure and discounting this value back to the present date. Chapter 12 covers the simulation of several types of stochastic processes under the risk-neutral measure.

Finally, this chapter introduces symbolic computation with `SymPy`. For a number of mathematical operations, like integration, differentiation, or the solving of equations, symbolic computation can prove a useful and efficient tool.

Further Resources

For further information on the Python libraries used in this chapter, consult the following web resources:

- See the `NumPy` Reference for details on the `NumPy` functions used in this chapter.
- Visit the `SciPy` documentation on optimization and root finding for details on `scipy.optimize`.
- Integration with `scipy.integrate` is explained in "Integration and ODEs".
- The `SymPy` website provides a wealth of examples and detailed documentation.

Chapter 13. Statistics

I can prove anything by statistics except the truth.

—George Canning

Statistics is a vast field, but the tools and results it provides have become indispensable for finance. This explains the popularity of domain-specific languages like R in the finance industry. The more elaborate and complex statistical models become, the more important it is to have available easy-to-use and high-performing computational solutions.

A single chapter in a book like this one cannot do justice to the richness and depth of the field of statistics. Therefore, the approach—as in many other chapters—is to focus on selected topics that seem of importance or that provide a good starting point when it comes to the use of Python for the particular tasks at hand. The chapter has four focal points:

"Normality Tests"

A large number of important financial models, like modern or mean-variance portfolio theory (MPT) and the capital asset pricing model (CAPM), rest on the assumption that returns of securities are normally distributed. Therefore, this chapter presents approaches to test a given time series for normality of returns.

"Portfolio Optimization"

MPT can be considered one of the biggest successes of statistics in finance. Starting in the early 1950s with the work of pioneer Harry Markowitz, this theory began to replace people's reliance on judgment and experience with rigorous mathematical and statistical methods when it comes to the investment of money in financial markets. In that sense, it is maybe the first real quantitative model and approach in finance.

"Bayesian Statistics"

On a conceptual level, Bayesian statistics introduces the notion of *beliefs* of agents and the *updating of beliefs* to statistics. When it comes to linear

regression, for example, this might take the form of having a statistical distribution for regression parameters instead of single point estimates (e.g., for the intercept and slope of the regression line). Nowadays, Bayesian methods are widely used in finance, which is why this section illustrates Bayesian methods based on some examples.

"Machine Learning"

Machine learning (or statistical learning) is based on advanced statistical methods and is considered a subdiscipline of artificial intelligence (AI). Like statistics itself, machine learning offers a rich set of approaches and models to learn from data sets and create predictions based on what is learned. Different algorithms of learning are distinguished, such as those for *supervised learning* or *unsupervised learning*. The types of problems solved by the algorithms differ as well, such as *estimation* or *classification*. The examples presented in this chapter fall in the category of *supervised learning for classification*.

Many aspects in this chapter relate to date and/or time information. Refer to Appendix A for an overview of handling such data with Python, `NumPy`, and `pandas`.

Normality Tests

The *normal distribution* can be considered the most important distribution in finance and one of the major statistical building blocks of financial theory. Among others, the following cornerstones of financial theory rest to a large extent on the assumption that returns of a financial instrument are normally distributed:[1]

Portfolio theory

When stock returns are normally distributed, optimal portfolio choice can be cast into a setting where only the (expected) *mean return* and the *variance of the returns* (or the volatility) as well as the *covariances* between different stocks are relevant for an investment decision (i.e., an optimal portfolio composition).

Capital asset pricing model

Again, when stock returns are normally distributed, prices of single stocks can be elegantly expressed in linear relationship to a broad market index; the relationship is generally expressed by a measure for the co-movement of a single stock with the market index called beta or β.

Efficient markets hypothesis

An *efficient* market is a market where prices reflect all available information, where "all" can be defined more narrowly or more widely (e.g., as in "all publicly available" information vs. including also "only privately available" information). If this hypothesis holds true, then stock prices fluctuate randomly and returns are normally distributed.

Option pricing theory

Brownian motion is *the* benchmark model for the modeling of random price movements of financial instruments; the famous Black-Scholes-Merton option pricing formula uses a geometric Brownian motion as the model for a stock's random price fluctuations over time, leading to log-normally distributed prices and normally distributed returns.

This by far nonexhaustive list underpins the importance of the normality assumption in finance.

Benchmark Case

To set the stage for further analyses, the analysis starts with the geometric Brownian motion as one of the canonical stochastic processes used in financial modeling. The following can be said about the characteristics of paths from a geometric Brownian motion *S*:

Normal log returns

Log returns $\log \frac{S_t}{S_s} = \log S_t - \log S_s$ between two times $0 < s < t$ are *normally* distributed.

Log-normal values

At any time $t > 0$, the values S_t are *log-normally* distributed.

Figure 13-6 also supports the hypothesis that the log index levels are normally distributed:

```
In [26]: sm.qqplot(log_data, line='s')
         plt.xlabel('theoretical quantiles')
         plt.ylabel('sample quantiles');
```

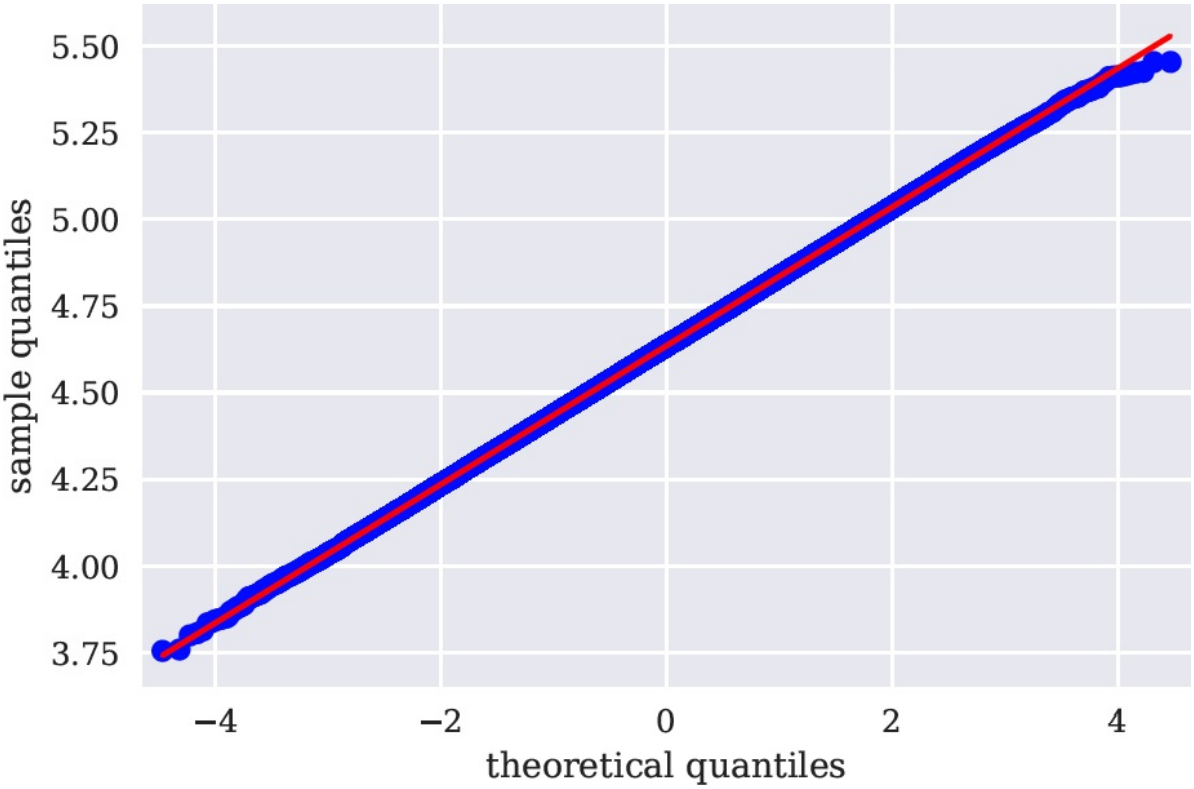

Figure 13-6. Quantile-quantile plot for log index levels of geometric Brownian motion

> **NORMALITY**
>
> The normality assumption with regard to the uncertain returns of financial instruments is central to a number of financial theories. Python provides efficient statistical and graphical means to test whether time series data is normally distributed or not.

Real-World Data

This section analyzes four historical financial time series, two for technology stocks and two for exchange traded funds (ETFs):

- `APPL.O`: Apple Inc. stock price

- `MSFT.O`: Microsoft Inc. stock price
- `SPY`: SPDR S&P 500 ETF Trust
- `GLD`: SPDR Gold Trust

The data management tool of choice is `pandas` (see Chapter 8). Figure 13-7 shows the normalized prices over time:

```
In [27]: import pandas as pd

In [28]: raw = pd.read_csv('../../source/tr_eikon_eod_data.csv',
                           index_col=0, parse_dates=True).dropna()

In [29]: symbols = ['SPY', 'GLD', 'AAPL.O', 'MSFT.O']

In [30]: data = raw[symbols]
         data = data.dropna()

In [31]: data.info()
         <class 'pandas.core.frame.DataFrame'>
         DatetimeIndex: 2138 entries, 2010-01-04 to 2018-06-29
         Data columns (total 4 columns):
         SPY       2138 non-null float64
         GLD       2138 non-null float64
         AAPL.O    2138 non-null float64
         MSFT.O    2138 non-null float64
         dtypes: float64(4)
         memory usage: 83.5 KB

In [32]: data.head()
Out[32]:                SPY     GLD     AAPL.O  MSFT.O
         Date
         2010-01-04  113.33  109.80  30.572827  30.950
         2010-01-05  113.63  109.70  30.625684  30.960
         2010-01-06  113.71  111.51  30.138541  30.770
         2010-01-07  114.19  110.82  30.082827  30.452
         2010-01-08  114.57  111.37  30.282827  30.660

In [33]: (data / data.iloc[0] * 100).plot(figsize=(10, 6))
```

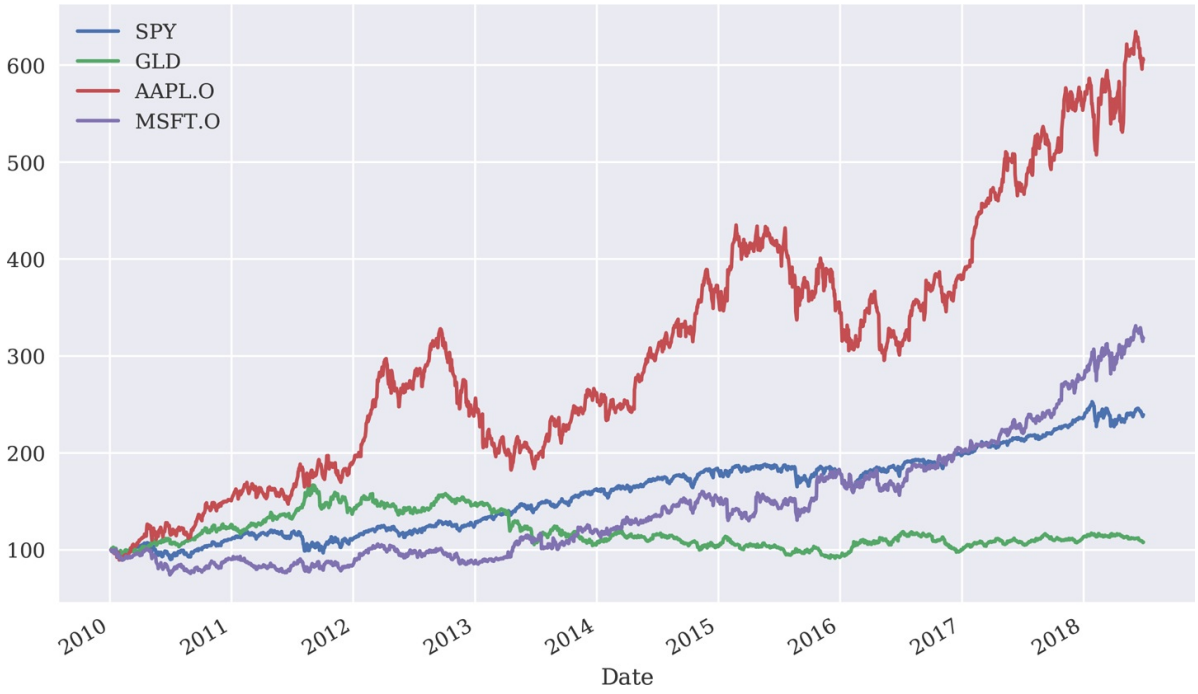

Figure 13-7. Normalized prices of financial instruments over time

Figure 13-8 shows the log returns of the financial instruments as histograms:

```
In [34]: log_returns = np.log(data / data.shift(1))
         log_returns.head()
Out[34]:                  SPY       GLD    AAPL.O    MSFT.O
         Date
         2010-01-04       NaN       NaN       NaN       NaN
         2010-01-05  0.002644 -0.000911  0.001727  0.000323
         2010-01-06  0.000704  0.016365 -0.016034 -0.006156
         2010-01-07  0.004212 -0.006207 -0.001850 -0.010389
         2010-01-08  0.003322  0.004951  0.006626  0.006807

In [35]: log_returns.hist(bins=50, figsize=(10, 8));
```

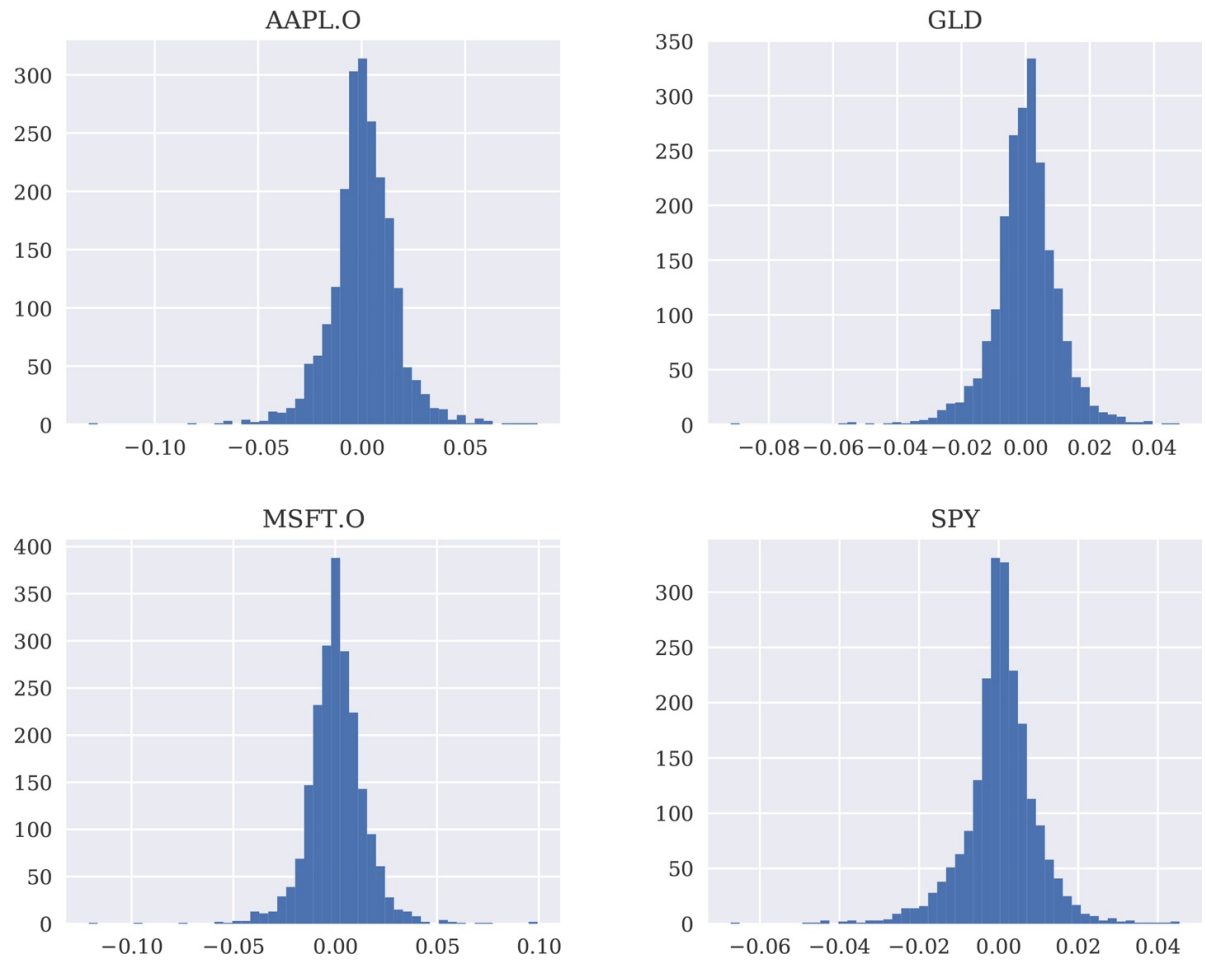

Figure 13-8. Histograms of log returns for financial instruments

As a next step, consider the different statistics for the time series data sets. The kurtosis values seem to be especially far from normal for all four data sets:

```
In [36]: for sym in symbols:
             print('\nResults for symbol {}'.format(sym))
             print(30 * '-')
             log_data = np.array(log_returns[sym].dropna())
             print_statistics(log_data)  ❶

         Results for symbol SPY
         ------------------------------
              statistic           value
         ------------------------------
                   size      2137.00000
                    min        -0.06734
                    max         0.04545
                   mean         0.00041
                    std         0.00933
                   skew        -0.52189
```

```
      kurtosis         4.52432

Results for symbol GLD
------------------------------
     statistic           value
------------------------------
          size      2137.00000
           min        -0.09191
           max         0.04795
          mean         0.00004
           std         0.01020
          skew        -0.59934
      kurtosis         5.68423

Results for symbol AAPL.O
------------------------------
     statistic           value
------------------------------
          size      2137.00000
           min        -0.13187
           max         0.08502
          mean         0.00084
           std         0.01591
          skew        -0.23510
      kurtosis         4.78964

Results for symbol MSFT.O
------------------------------
     statistic           value
------------------------------
          size      2137.00000
           min        -0.12103
           max         0.09941
          mean         0.00054
           std         0.01421
          skew        -0.09117
      kurtosis         7.29106
```

❶ Statistics for time series of financial instruments.

Figure 13-9 shows the QQ plot for the SPY ETF. Obviously, the sample quantile values do not lie on a straight line, indicating "non-normality." On the left and right sides there are many values that lie well below the line and well above the line, respectively. In other words, the time series data exhibits *fat tails*. This term refers to a (frequency) distribution where large negative and positive values are observed more often than a normal distribution would imply. The same

conclusions can be drawn from Figure 13-10, which presents the data for the Microsoft stock. There also seems to be evidence for a fat-tailed distribution:

```
In [37]: sm.qqplot(log_returns['SPY'].dropna(), line='s')
         plt.title('SPY')
         plt.xlabel('theoretical quantiles')
         plt.ylabel('sample quantiles');
In [38]: sm.qqplot(log_returns['MSFT.O'].dropna(), line='s')
         plt.title('MSFT.O')
         plt.xlabel('theoretical quantiles')
         plt.ylabel('sample quantiles');
```

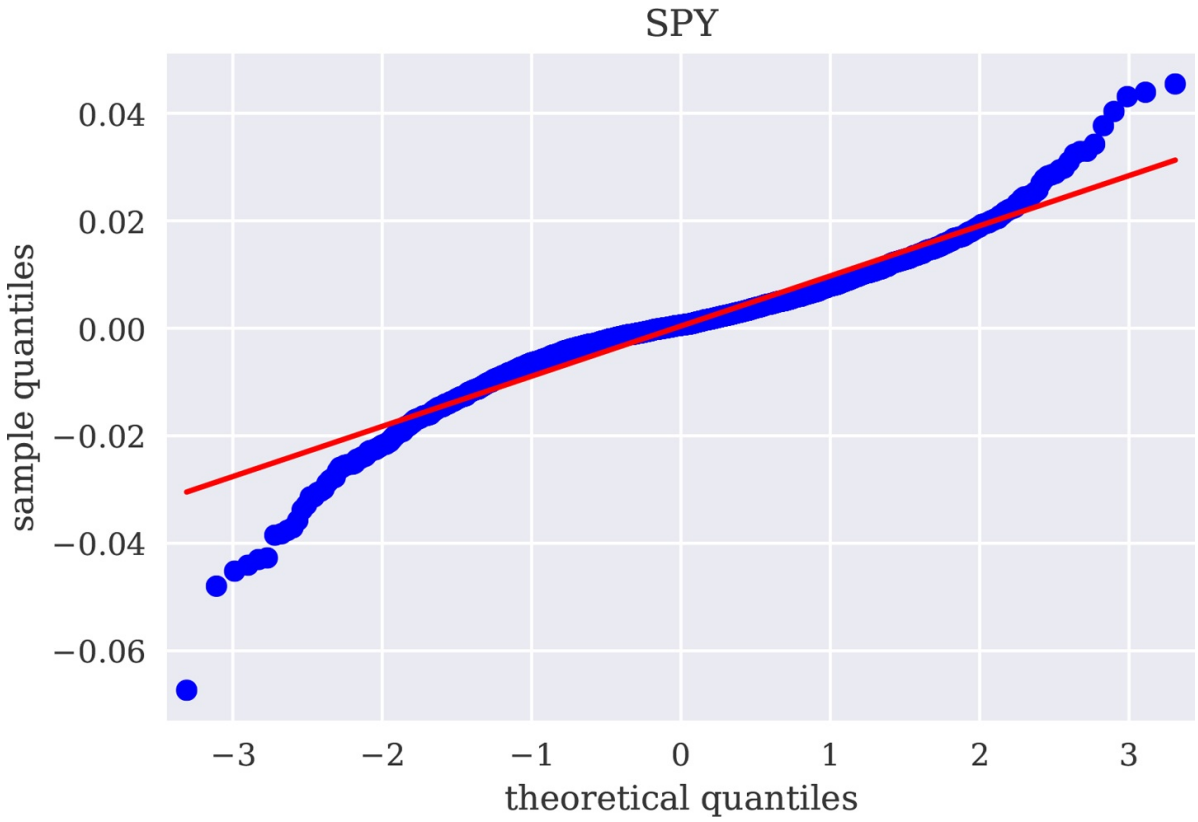

Figure 13-9. Quantile-quantile plot for SPY log returns

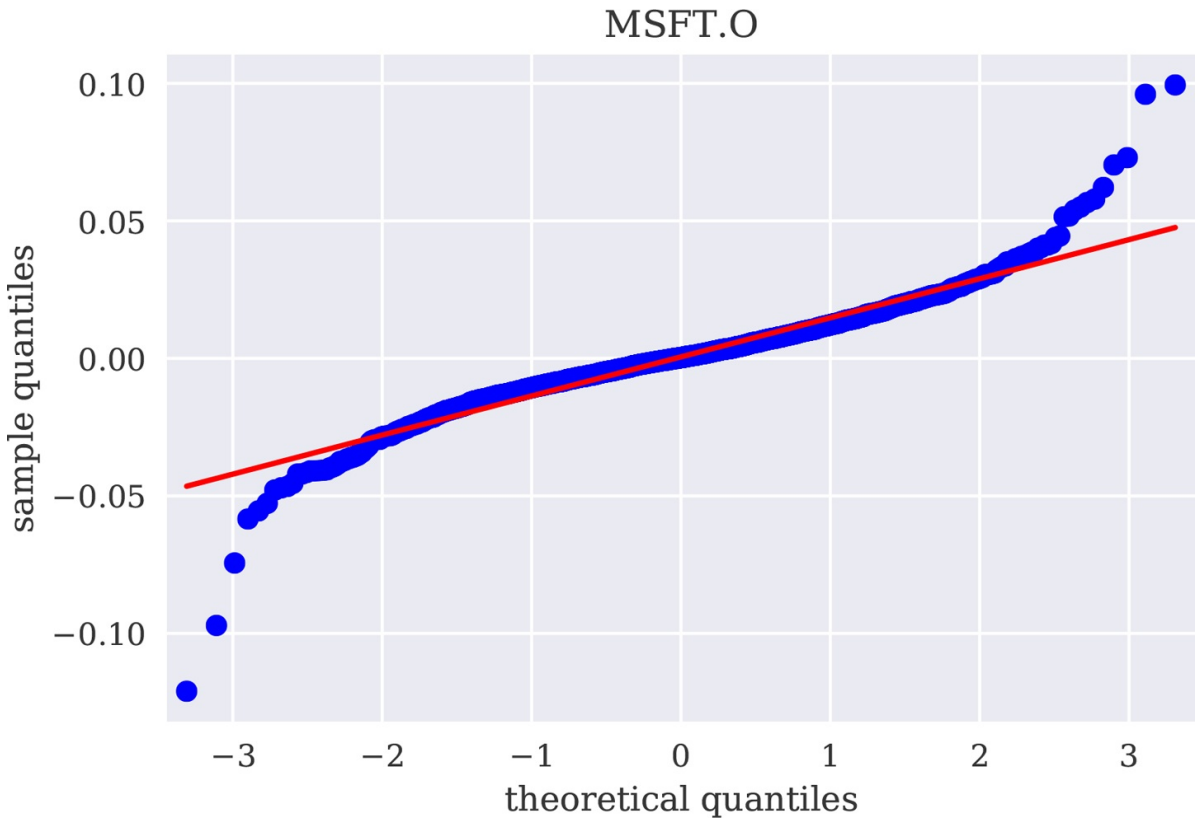

Figure 13-10. Quantile-quantile plot for MSFT.O log returns

This finally leads to the statistical normality tests:

```
In [39]: for sym in symbols:
             print('\nResults for symbol {}'.format(sym))
             print(32 * '-')
             log_data = np.array(log_returns[sym].dropna())
             normality_tests(log_data)  ❶

         Results for symbol SPY
         --------------------------------
         Skew of data set          -0.522
         Skew test p-value          0.000
         Kurt of data set           4.524
         Kurt test p-value          0.000
         Norm test p-value          0.000

         Results for symbol GLD
         --------------------------------
         Skew of data set          -0.599
         Skew test p-value          0.000
         Kurt of data set           5.684
         Kurt test p-value          0.000
         Norm test p-value          0.000
```

```
Results for symbol AAPL.O
--------------------------------
Skew of data set          -0.235
Skew test p-value          0.000
Kurt of data set           4.790
Kurt test p-value          0.000
Norm test p-value          0.000

Results for symbol MSFT.O
--------------------------------
Skew of data set          -0.091
Skew test p-value          0.085
Kurt of data set           7.291
Kurt test p-value          0.000
Norm test p-value          0.000
```

❶ Normality test results for the times series of the financial instruments.

The *p*-values of the different tests are all zero, *strongly rejecting the test hypothesis* that the different sample data sets are normally distributed. This shows that the normal assumption for stock market returns and other asset classes—as, for example, embodied in the geometric Brownian motion model—cannot be justified in general and that one might have to use richer models that are able to generate fat tails (e.g., jump diffusion models or models with stochastic volatility).

Portfolio Optimization

Modern or mean-variance portfolio theory is a major cornerstone of financial theory. Based on this theoretical breakthrough the Nobel Prize in Economics was awarded to its inventor, Harry Markowitz, in 1990. Although formulated in the 1950s, it is still a theory taught to finance students and applied in practice today (often with some minor or major modifications).[3] This section illustrates the fundamental principles of the theory.

Chapter 5 in the book by Copeland, Weston, and Shastri (2005) provides an introduction to the formal topics associated with MPT. As pointed out previously, the assumption of normally distributed returns is fundamental to the theory:

By looking only at mean and variance, we are necessarily assuming that no other statistics are necessary to describe the distribution of end-of-period wealth. Unless investors have a special type of utility function (quadratic utility function), it is necessary to assume that returns have a normal distribution, which can be completely described by mean and variance.

The Data

The analysis and examples that follow use the same financial instruments as before. The basic idea of MPT is to make use of *diversification* to achieve a minimal portfolio risk given a target return level or a maximum portfolio return given a certain level of risk. One would expect such diversification effects for the right combination of a larger number of assets and a certain diversity in the assets. However, to convey the basic ideas and to show typical effects, four financial instruments shall suffice. Figure 13-11 shows the frequency distribution of the log returns for the financial instruments:

```
In [40]: symbols = ['AAPL.O', 'MSFT.O', 'SPY', 'GLD']  ❶

In [41]: noa = len(symbols)  ❷

In [42]: data = raw[symbols]

In [43]: rets = np.log(data / data.shift(1))

In [44]: rets.hist(bins=40, figsize=(10, 8));
```

❶ Four financial instruments for portfolio composition.

❷ Number of financial instruments defined.

The *covariance matrix* for the financial instruments to be invested in is the central piece of the portfolio selection process. `pandas` has a built-in method to generate the covariance matrix on which the same scaling factor is applied:

```
In [45]: rets.mean() * 252  ❶
Out[45]: AAPL.O    0.212359
         MSFT.O    0.136648
         SPY       0.102928
         GLD       0.009141
         dtype: float64
```

```
In [46]: rets.cov() * 252  ❷
Out[46]:           AAPL.O    MSFT.O       SPY       GLD
         AAPL.O  0.063773  0.023427  0.021039  0.001513
         MSFT.O  0.023427  0.050917  0.022244 -0.000347
         SPY     0.021039  0.022244  0.021939  0.000062
         GLD     0.001513 -0.000347  0.000062  0.026209
```

❶ Annualized mean returns.

❷ Annualized covariance matrix.

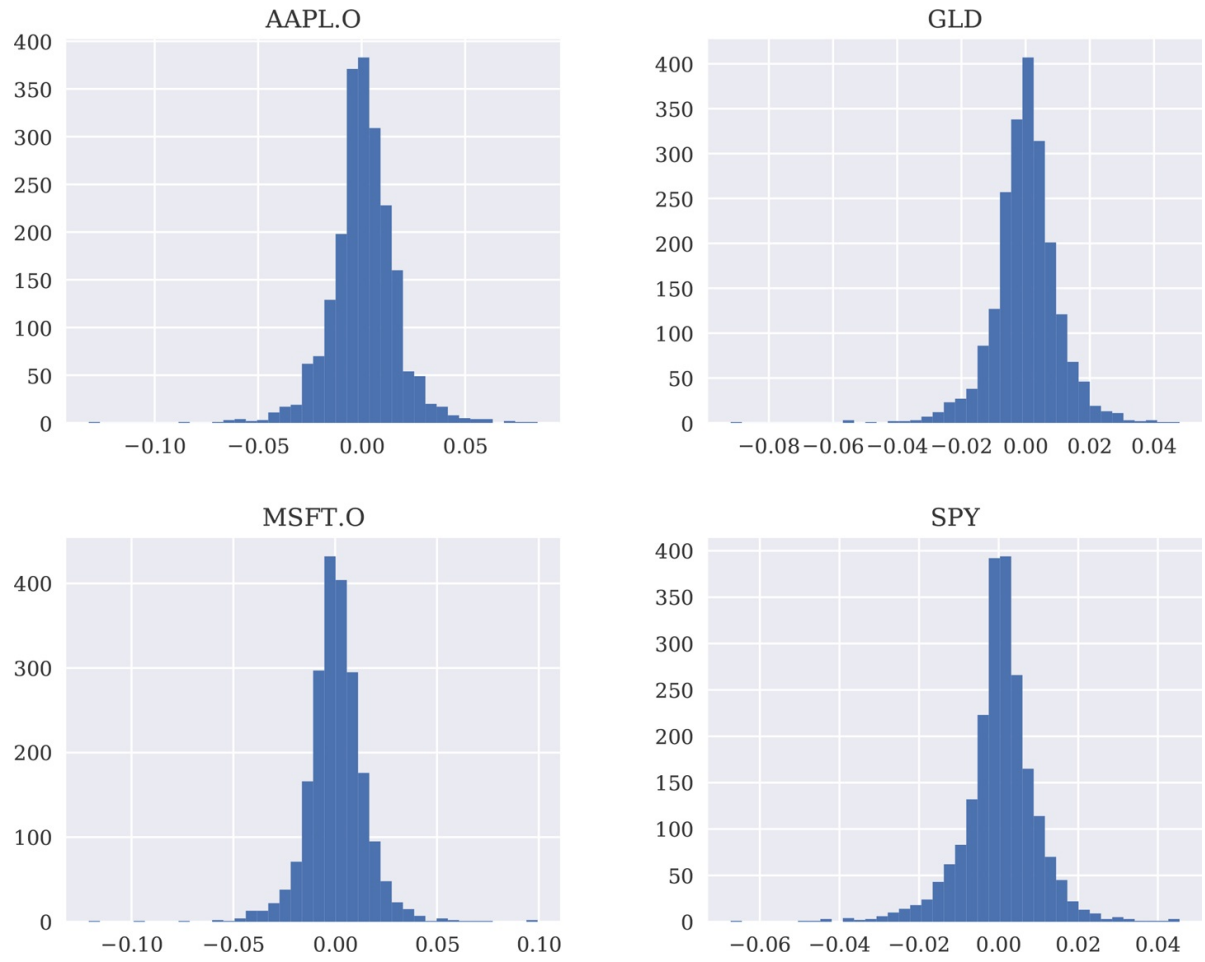

Figure 13-11. Histograms of log returns of financial instruments

The Basic Theory

In what follows, it is assumed that an investor is not allowed to set up short positions in a financial instrument. Only long positions are allowed, which implies that 100% of the investor's wealth has to be divided among the available instruments in such a way that all positions are long (positive) *and* that the

positions add up to 100%. Given the four instruments, one could, for example, invest equal amounts into every such instrument—i.e., 25% of the available wealth in each. The following code generates four uniformly distributed random numbers between 0 and 1 and then normalizes the values such that the sum of all values equals 1:

```
In [47]: weights = np.random.random(noa)  ❶
         weights /= np.sum(weights)  ❷

In [48]: weights
Out[48]: array([0.07650728, 0.06021919, 0.63364218, 0.22963135])

In [49]: weights.sum()
Out[49]: 1.0
```

❶ Random portfolio weights …

❷ … normalized to 1 or 100%.

As verified here, the weights indeed add up to 1; i.e., $\sum_I w_i = 1$, where *I* is the number of financial instruments and $w_i > 0$ is the weight of financial instrument *i*. Equation 13-1 provides the formula for the *expected portfolio return* given the weights for the single instruments. This is an *expected* portfolio return in the sense that historical mean performance is assumed to be the best estimator for future (expected) performance. Here, the r_i are the state-dependent future returns (vector with return values assumed to be normally distributed) and μ_i is the expected return for instrument *i*. Finally, w^T is the transpose of the weights vector and μ is the vector of the expected security returns.

Equation 13-1. General formula for expected portfolio return

$$\begin{aligned} \mu_p &= \mathbf{E}\left(\sum_I w_i r_i\right) \\ &= \sum_I w_i \mathbf{E}\left(r_i\right) \\ &= \sum_I w_i \mu_i \\ &= w^T \mu \end{aligned}$$

Translated into Python this boils down to a single line of code including annualization:

```
In [50]: np.sum(rets.mean() * weights) * 252  ❶
Out[50]: 0.09179459482057793
```

❶ Annualized portfolio return given the portfolio weights.

The second object of importance in MPT is the *expected portfolio variance*. The covariance between two securities is defined by $\sigma_{ij} = \sigma_{ji} = \mathbf{E}\left(r_i - \mu_i\right)\left(r_j - \mu_j\right)$. The variance of a security is the special case of the covariance with itself: $\sigma_i^2 = \mathbf{E}\left(\left(r_i - \mu_i\right)^2\right)$. Equation 13-2 provides the covariance matrix for a portfolio of securities (assuming an equal weight of 1 for every security).

Equation 13-2. Portfolio covariance matrix

$$\Sigma = \begin{bmatrix} \sigma_1^2 & \sigma_{12} & \cdots & \sigma_{1I} \\ \sigma_{21} & \sigma_2^2 & \cdots & \sigma_{2I} \\ \vdots & \vdots & \ddots & \vdots \\ \sigma_{I1} & \sigma_{I2} & \cdots & \sigma_I^2 \end{bmatrix}$$

Equipped with the portfolio covariance matrix, Equation 13-3 then provides the formula for the expected portfolio variance.

Equation 13-3. General formula for expected portfolio variance

$$\begin{aligned} \sigma_p^2 &= \mathbf{E}\left(\left(r - \mu\right)^2\right) \\ &= \sum_{i \in I} \sum_{j \in I} w_i w_j \sigma_{ij} \\ &= w^T \Sigma w \end{aligned}$$

In Python, this all again boils down to a single line of code, making heavy use of `NumPy` vectorization capabilities. The `np.dot()` function gives the dot product of

two vectors/matrices. The `T` attribute or `transpose()` method gives the transpose of a vector or matrix. Given the portfolio variance, the (expected) portfolio standard deviation or volatility $\sigma_p = \sqrt{\sigma_p^2}$ is then only one square root away:

```
In [51]: np.dot(weights.T, np.dot(rets.cov() * 252, weights))  ❶
Out[51]: 0.014763288666485574

In [52]: math.sqrt(np.dot(weights.T, np.dot(rets.cov() * 252, weights)))  ❷
Out[52]: 0.12150427427249452
```

❶ Annualized *portfolio variance* given the portfolio weights.
❷ Annualized *portfolio volatility* given the portfolio weights.

> **PYTHON AND VECTORIZATION**
>
> The MPT example shows how efficient it is with Python to translate mathematical concepts, like portfolio return or portfolio variance, into executable, vectorized code (an argument made in Chapter 1).

This mainly completes the tool set for mean-variance portfolio selection. Of paramount interest to investors is what risk-return profiles are possible for a given set of financial instruments, and their statistical characteristics. To this end, the following implements a Monte Carlo simulation (see Chapter 12) to generate random portfolio weight vectors on a larger scale. For every simulated allocation, the code records the resulting expected portfolio return and variance. To simplify the code, two functions, `port_ret()` and `port_vol()`, are defined:

```
In [53]: def port_ret(weights):
             return np.sum(rets.mean() * weights) * 252

In [54]: def port_vol(weights):
             return np.sqrt(np.dot(weights.T, np.dot(rets.cov() * 252, weights)))

In [55]: prets = []
         pvols = []
         for p in range (2500):  ❶
             weights = np.random.random(noa)  ❶
             weights /= np.sum(weights)  ❶
```

```
        prets.append(port_ret(weights))  ❷
        pvols.append(port_vol(weights))  ❷
prets = np.array(prets)
pvols = np.array(pvols)
```

❶ Monte Carlo simulation of portfolio weights.

❷ Collects the resulting statistics in `list` objects.

Figure 13-12 illustrates the results of the Monte Carlo simulation. In addition, it provides results for the Sharpe ratio, defined as $SR \equiv \frac{\mu_p - r_f}{\sigma_p}$—i.e., the expected excess return of the portfolio over the risk-free short rate r_f divided by the expected standard deviation of the portfolio. For simplicity, $r_f \equiv 0$ is assumed:

```
In [56]: plt.figure(figsize=(10, 6))
         plt.scatter(pvols, prets, c=prets / pvols,
                     marker='o', cmap='coolwarm')
         plt.xlabel('expected volatility')
         plt.ylabel('expected return')
         plt.colorbar(label='Sharpe ratio');
```

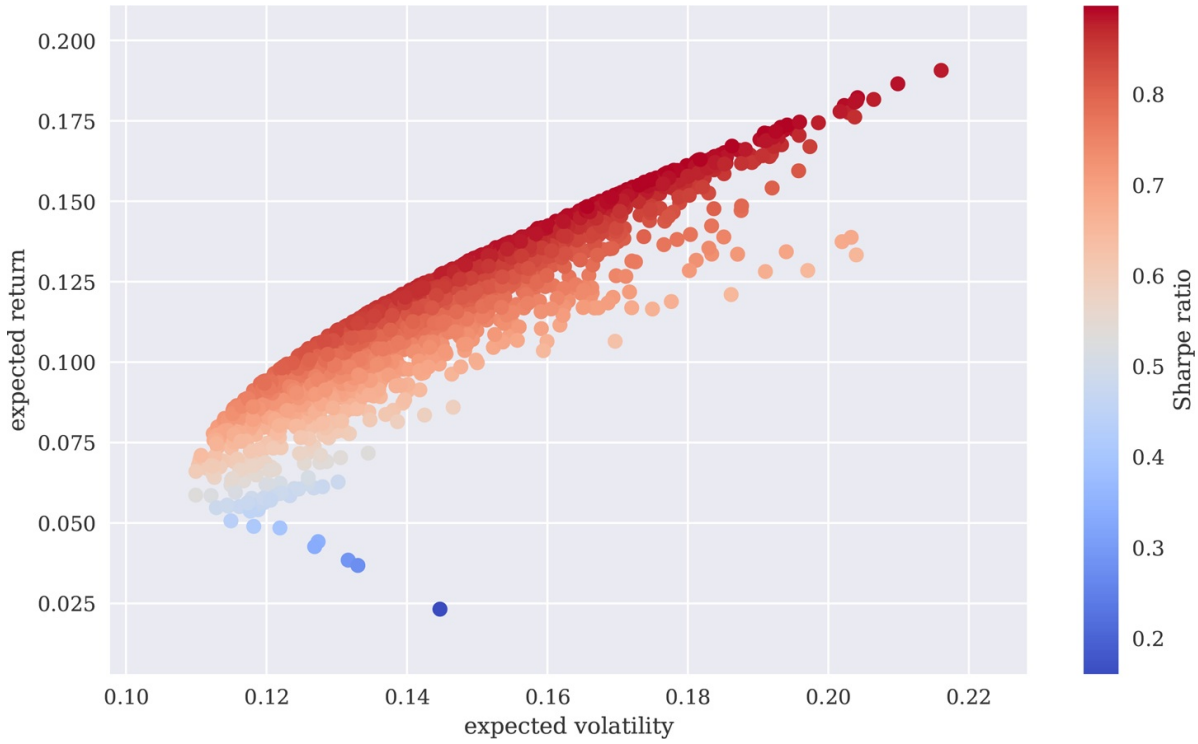

Figure 13-12. Expected return and volatility for random portfolio weights

It is clear by inspection of Figure 13-12 that not all weight distributions perform well when measured in terms of mean and volatility. For example, for a fixed risk level of, say, 15%, there are multiple portfolios that all show different returns. As an investor, one is generally interested in the maximum return given a fixed risk level or the minimum risk given a fixed return expectation. This set of portfolios then makes up the so-called *efficient frontier*. This is derived later in this section.

Optimal Portfolios

This *minimization* function is quite general and allows for equality constraints, inequality constraints, and numerical bounds for the parameters.

First, the *maximization of the Sharpe ratio*. Formally, the negative value of the Sharpe ratio is minimized to derive at the maximum value and the optimal portfolio composition. The constraint is that all parameters (weights) add up to 1. This can be formulated as follows using the conventions of the `minimize()` function.[4] The parameter values (weights) are also bound to be between 0 and 1. These values are provided to the minimization function as a tuple of tuples.

The only input that is missing for a call of the optimization function is a starting parameter list (initial guess for the weights vector). An equal distribution of weights will do:

```
In [57]: import scipy.optimize as sco

In [58]: def min_func_sharpe(weights):  ❶
             return -port_ret(weights) / port_vol(weights)  ❶

In [59]: cons = ({'type': 'eq', 'fun': lambda x:  np.sum(x) - 1})  ❷

In [60]: bnds = tuple((0, 1) for x in range(noa))  ❸

In [61]: eweights = np.array(noa * [1. / noa,])  ❹
         eweights  ❹
Out[61]: array([0.25, 0.25, 0.25, 0.25])

In [62]: min_func_sharpe(eweights)
Out[62]: -0.8436203363155397
```

❶ Function to be minimized.

❷ Equality constraint.
❸ Bounds for the parameters.
❹ Equal weights vector.

Calling the function returns more than just the optimal parameter values. The results are stored in an object called `opts`. The main interest lies in getting the optimal portfolio composition. To this end, one can access the results object by providing the key of interest; i.e., `x` in this case:

```
In [63]: %%time
         opts = sco.minimize(min_func_sharpe, eweights,
                             method='SLSQP', bounds=bnds,
                             constraints=cons)  ❶
         CPU times: user 67.6 ms, sys: 1.94 ms, total: 69.6 ms
         Wall time: 75.2 ms

In [64]: opts  ❷
Out[64]:      fun: -0.8976673894052725
         jac: array([ 8.96826386e-05,  8.30739737e-05, -2.45958567e-04,
          1.92895532e-05])
          message: 'Optimization terminated successfully.'
             nfev: 36
              nit: 6
             njev: 6
           status: 0
          success: True
                x: array([0.51191354, 0.19126414, 0.25454109, 0.04228123])

In [65]: opts['x'].round(3)  ❸
Out[65]: array([0.512, 0.191, 0.255, 0.042])

In [66]: port_ret(opts['x']).round(3)  ❹
Out[66]: 0.161

In [67]: port_vol(opts['x']).round(3)  ❺
Out[67]: 0.18

In [68]: port_ret(opts['x']) / port_vol(opts['x'])  ❻
Out[68]: 0.8976673894052725
```

❶ The optimization (i.e., minimization of function `min_func_sharpe()`).
❷ The results from the optimization.
❸ The optimal portfolio weights.
❹ The resulting portfolio return.

❺ The resulting portfolio volatility.

❻ The maximum Sharpe ratio.

Next, the *minimization of the variance* of the portfolio. This is the same as minimizing the volatility:

```
In [69]: optv = sco.minimize(port_vol, eweights,
                             method='SLSQP', bounds=bnds,
                             constraints=cons)  ❶

In [70]: optv
Out[70]:      fun: 0.1094215526341138
              jac: array([0.11098004, 0.10948556, 0.10939826, 0.10944918])
          message: 'Optimization terminated successfully.'
             nfev: 54
              nit: 9
             njev: 9
           status: 0
          success: True
            x: array([1.62630326e-18, 1.06170720e-03, 5.43263079e-01,
           4.55675214e-01])

In [71]: optv['x'].round(3)
Out[71]: array([0.   , 0.001, 0.543, 0.456])

In [72]: port_vol(optv['x']).round(3)
Out[72]: 0.109

In [73]: port_ret(optv['x']).round(3)
Out[73]: 0.06

In [74]: port_ret(optv['x']) / port_vol(optv['x'])
Out[74]: 0.5504173653075624
```

❶ The minimization of the portfolio volatility.

This time, the portfolio is made up of only three financial instruments. This portfolio mix leads to the so-called *minimum volatility* or *minimum variance portfolio*.

Efficient Frontier

The derivation of all optimal portfolios—i.e., all portfolios with minimum volatility for a given target return level (or all portfolios with maximum return for a given risk level)—is similar to the previous optimizations. The only

difference is that one has to iterate over multiple starting conditions.

The approach taken is to fix a target return level and to derive for each such level those portfolio weights that lead to the minimum volatility value. For the optimization, this leads to two conditions: one for the target return level, `tret`, and one for the sum of the portfolio weights as before. The boundary values for each parameter stay the same. When iterating over different target return levels (`trets`), one condition for the minimization changes. That is why the constraints dictionary is updated during every loop:

```
In [75]: cons = ({'type': 'eq', 'fun': lambda x:  port_ret(x) - tret},
                 {'type': 'eq', 'fun': lambda x:  np.sum(x) - 1})  ❶

In [76]: bnds = tuple((0, 1) for x in weights)

In [77]: %%time
         trets = np.linspace(0.05, 0.2, 50)
         tvols = []
         for tret in trets:
             res = sco.minimize(port_vol, eweights, method='SLSQP',
                                bounds=bnds, constraints=cons)  ❷
             tvols.append(res['fun'])
         tvols = np.array(tvols)
         CPU times: user 2.6 s, sys: 13.1 ms, total: 2.61 s
         Wall time: 2.66 s
```

❶ The two binding constraints for the efficient frontier.

❷ The minimization of portfolio volatility for different target returns.

Figure 13-13 shows the optimization results. Crosses indicate the optimal portfolios given a certain target return; the dots are, as before, the random portfolios. In addition, the figure shows two larger stars, one for the minimum volatility/variance portfolio (the leftmost portfolio) and one for the portfolio with the maximum Sharpe ratio:

```
In [78]: plt.figure(figsize=(10, 6))
         plt.scatter(pvols, prets, c=prets / pvols,
                     marker='.', alpha=0.8, cmap='coolwarm')
         plt.plot(tvols, trets, 'b', lw=4.0)
         plt.plot(port_vol(opts['x']), port_ret(opts['x']),
                  'y*', markersize=15.0)
         plt.plot(port_vol(optv['x']), port_ret(optv['x']),
                  'r*', markersize=15.0)
```

```
plt.xlabel('expected volatility')
plt.ylabel('expected return')
plt.colorbar(label='Sharpe ratio')
```

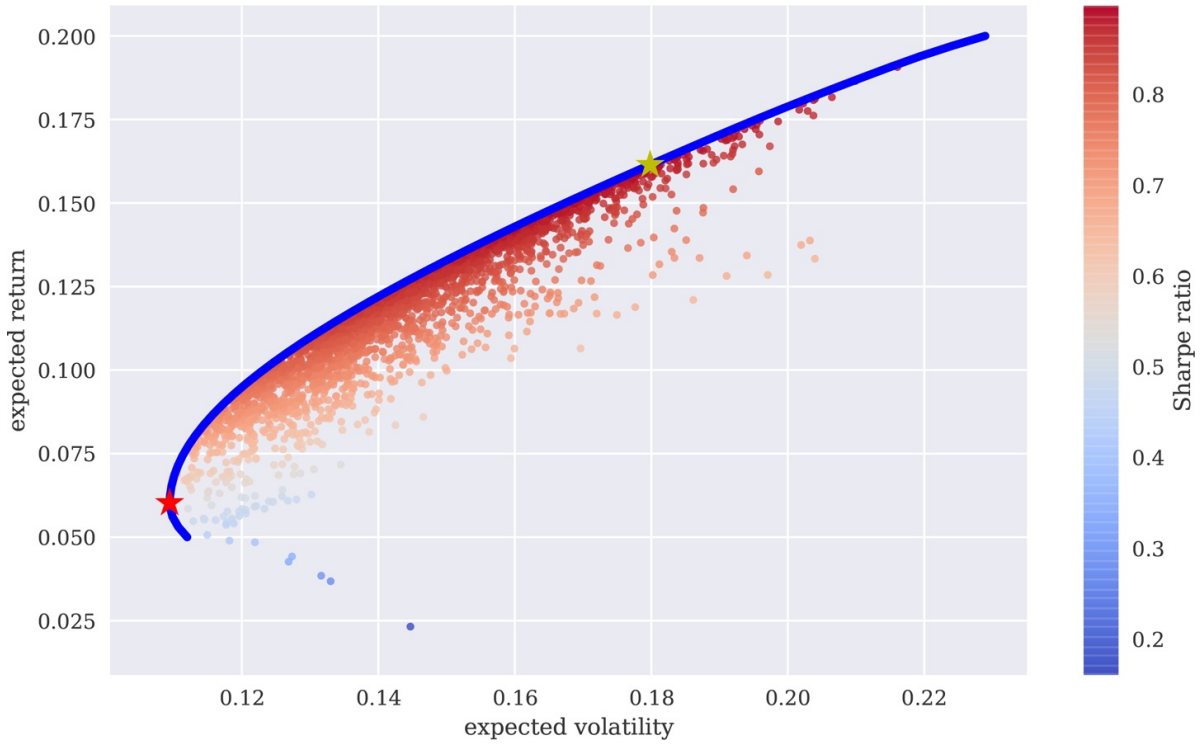

Figure 13-13. Minimum risk portfolios for given return levels (efficient frontier)

The *efficient frontier* is comprised of all optimal portfolios with a higher return than the absolute minimum variance portfolio. These portfolios dominate all other portfolios in terms of expected returns given a certain risk level.

Capital Market Line

In addition to risky financial instruments like stocks or commodities (such as gold), there is in general one universal, riskless investment opportunity available: *cash* or *cash accounts*. In an idealized world, money held in a cash account with a large bank can be considered riskless (e.g., through public deposit insurance schemes). The downside is that such a riskless investment generally yields only a small return, sometimes close to zero.

However, taking into account such a riskless asset enhances the efficient investment opportunity set for investors considerably. The basic idea is that investors first determine an efficient portfolio of risky assets and then add the

```
    print('{:>8s} | {:8.3f}'.format(kernel, acc))
  kernel | accuracy
--------------------
  linear |    0.848
    poly |    0.758
     rbf |    0.788
 sigmoid |    0.455
```

Conclusion

Statistics is not only an important discipline in its own right, but also provides indispensable tools for many other disciplines, like finance and the social sciences. It is impossible to give a broad overview of such a large subject in a single chapter. This chapter therefore focuses on four important topics, illustrating the use of Python and several statistics libraries on the basis of realistic examples:

Normality

> The normality assumption with regard to financial market returns is an important one for many financial theories and applications; it is therefore important to be able to test whether certain time series data conforms to this assumption. As seen in "Normality Tests"—via graphical and statistical means—real-world return data generally is *not* normally distributed.

Portfolio optimization

> MPT, with its focus on the mean and variance/volatility of returns, can be considered not only one of the first but also one of the major conceptual successes of statistics in finance; the important concept of investment *diversification* is beautifully illustrated in this context.

Bayesian statistics

> Bayesian statistics in general (and Bayesian regression in particular) has become a popular tool in finance, since this approach overcomes some shortcomings of other approaches, as introduced, for instance, in Chapter 11; even if the mathematics and the formalism are more involved, the fundamental ideas—like the updating of probability/distribution beliefs over time—are easily grasped (at least intuitively).

Machine learning

> Nowadays, machine learning has established itself in the financial domain alongside traditional statistical methods and techniques. The chapter introduces ML algorithms for unsupervised learning (such as *k*-means clustering) and supervised learning (such as DNN classifiers) and illustrates selected related topics, such as feature transforms and train-test splits.

Further Resources

For more information on the topics and packages covered in this chapter, consult the following online resources:

- The documentation on `SciPy`'s statistical functions
- The documentation of the `statsmodels` library
- Details on the optimization functions used in this chapter
- The documentation for `PyMC3`
- The documentation for `scikit-learn`

Useful references in book form for more background information are:

- Albon, Chris (2018). *Machine Learning with Python Cookbook*. Sebastopol, CA: O'Reilly.
- Alpaydin, Ethem (2016). *Machine Learning*. Cambridge, MA: MIT Press.
- Copeland, Thomas, Fred Weston, and Kuldeep Shastri (2005). *Financial Theory and Corporate Policy*. Boston, MA: Pearson.
- Downey, Allen (2013). *Think Bayes*. Sebastopol, CA: O'Reilly.
- Geweke, John (2005). *Contemporary Bayesian Econometrics and Statistics*. Hoboken, NJ: John Wiley & Sons.
- Hastie, Trevor, Robert Tibshirani, and Jerome Friedman (2009). *The Elements of Statistical Learning: Data Mining, Inference, and*

Part IV. Algorithmic Trading

This part of the book is about the use of Python for algorithmic trading. More and more trading platforms and brokers allow their clients to use, for example, REST APIs to programmatically retrieve historical data or streaming data, or to place buy and sell orders. What has been the domain of large financial institutions for a long period now has become accessible even to retail algorithmic traders. In this space, Python has secured a top position as a programming language and technology platform. Among other factors, this is driven by the fact that many trading platforms, such as the one from FXCM Forex Capital Markets, provide easy-to-use Python wrapper packages for their REST APIs.

This part of the book comprises three chapters:

- Chapter 14 introduces the FXCM trading platform, its REST API, and the `fxcmpy` wrapper package.
- Chapter 15 focuses on the use of methods from statistics and machine learning to derive algorithmic trading strategies; the chapter also shows how to use vectorized backtesting.
- Chapter 16 looks at the deployment of automated algorithmic trading strategies; it addresses capital management, backtesting for performance and risk, online algorithms, and deployment.

Chapter 15. Trading Strategies

[T]hey were silly enough to think you can look at the past to predict the future.

—The Economist[1]

This chapter is about the vectorized backtesting of algorithmic trading strategies. The term *algorithmic trading strategy* is used to describe any type of financial trading strategy that is based on an algorithm designed to take long, short, or neutral positions in financial instruments on its own without human interference. A simple algorithm, such as "altering every five minutes between a long and a neutral position in the stock of Apple, Inc.," satisfies this definition. For the purposes of this chapter and a bit more technically, an algorithmic trading strategy is represented by some Python code that, given the availability of new data, decides whether to buy or sell a financial instrument in order to take long, short, or neutral positions in it.

The chapter does not provide an overview of algorithmic trading strategies (see "Further Resources" for references that cover algorithmic trading strategies in more detail). It rather focuses on the technical aspects of the *vectorized backtesting* approach for a select few such strategies. With this approach the financial data on which the strategy is tested is manipulated in general as a whole, applying vectorized operations on `NumPy` `ndarray` and `pandas` `DataFrame` objects that store the financial data.[2]

Another focus of the chapter is the application of *machine and deep learning algorithms* to formulate algorithmic trading strategies. To this end, classification algorithms are trained on historical data in order to predict future directional market movements. This in general requires the transformation of the financial data from real values to a relatively small number of categorical values.[3] This allows us to harness the pattern recognition power of such algorithms.

The chapter is broken down into the following sections:

"Simple Moving Averages"

> This section focuses on an algorithmic trading strategy based on simple moving averages and how to backtest such a strategy.

“Random Walk Hypothesis”

This section introduces the random walk hypothesis.

“Linear OLS Regression”

This section looks at using OLS regression to derive an algorithmic trading strategy.

“Clustering”

In this section, we explore using unsupervised learning algorithms to derive algorithmic trading strategies.

“Frequency Approach”

This section introduces a simple frequentist approach for algorithmic trading.

“Classification”

Here we look at classification algorithms from machine learning for algorithmic trading.

“Deep Neural Networks”

This section focuses on deep neural networks and how to use them for algorithmic trading.

Simple Moving Averages

Trading based on simple moving averages (SMAs) is a decades-old trading approach (see, for example, the paper by Brock et al. (1992)). Although many traders use SMAs for their discretionary trading, they can also be used to formulate simple algorithmic trading strategies. This section uses SMAs to introduce vectorized backtesting of algorithmic trading strategies. It builds on the technical analysis example in Chapter 8.

Data Import

First, some imports:

```
In [1]: import numpy as np
        import pandas as pd
        import datetime as dt
        from pylab import mpl, plt

In [2]: plt.style.use('seaborn')
        mpl.rcParams['font.family'] = 'serif'
        %matplotlib inline
```

Second, the reading of the raw data and the selection of the financial time series for a single symbol, the stock of Apple, Inc. (`AAPL.O`). The analysis in this section is based on end-of-day data; intraday data is used in subsequent sections:

```
In [3]: raw = pd.read_csv('../../source/tr_eikon_eod_data.csv',
                          index_col=0, parse_dates=True)

In [4]: raw.info()
        <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 [5]: symbol = 'AAPL.O'

In [6]: data = (
            pd.DataFrame(raw[symbol])
            .dropna()
        )
```

Trading Strategy

Third, the calculation of the SMA values for two different rolling window sizes. Figure 15-1 shows the three time series visually:

```
In [7]: SMA1 = 42  ❶
        SMA2 = 252  ❷

In [8]: data['SMA1'] = data[symbol].rolling(SMA1).mean()  ❶
        data['SMA2'] = data[symbol].rolling(SMA2).mean()  ❷

In [9]: data.plot(figsize=(10, 6));
```

❶ Calculates the values for the *shorter* SMA.

❷ Calculates the values for the *longer* SMA.

Figure 15-1. Apple stock price and two simple moving averages

Fourth, the derivation of the positions. The trading rules are:

- Go *long* (= `+1`) when the shorter SMA is above the longer SMA.
- Go *short* (= `-1`) when the shorter SMA is below the longer SMA.[4]

The positions are visualized in Figure 15-2:

```
In [10]: data.dropna(inplace=True)

In [11]: data['Position'] = np.where(data['SMA1'] > data['SMA2'], 1, -1)  ❶

In [12]: data.tail()
Out[12]:              AAPL.O        SMA1        SMA2  Position
```

```
Date
2018-06-25  182.17  185.606190  168.265556         1
2018-06-26  184.43  186.087381  168.418770         1
2018-06-27  184.16  186.607381  168.579206         1
2018-06-28  185.50  187.089286  168.736627         1
2018-06-29  185.11  187.470476  168.901032         1
```

```
In [13]: ax = data.plot(secondary_y='Position', figsize=(10, 6))
         ax.get_legend().set_bbox_to_anchor((0.25, 0.85));
```

❶ `np.where(cond, a, b)` evaluates the condition `cond` element-wise and places `a` when `True` and `b` otherwise.

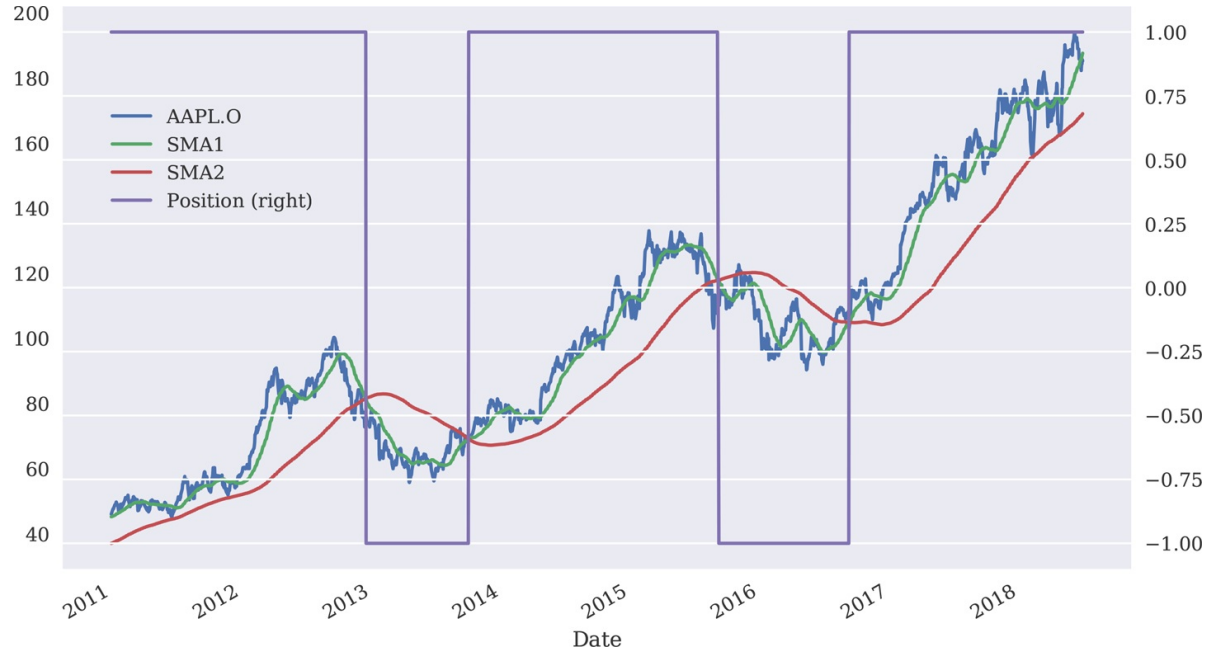

Figure 15-2. Apple stock price, two SMAs, and resulting positions

This replicates the results derived in Chapter 8. What is not addressed there is if following the trading rules—i.e., implementing the algorithmic trading strategy—is superior compared to the benchmark case of simply going long on the Apple stock over the whole period. Given that the strategy leads to two periods only during which the Apple stock should be shorted, differences in the performance can only result from these two periods.

Vectorized Backtesting

The vectorized backtesting can now be implemented as follows. First, the log returns are calculated. Then the positionings, represented as `+1` or `-1`, are

multiplied by the relevant log return. This simple calculation is possible since a long position earns the return of the Apple stock and a short position earns the negative return of the Apple stock. Finally, the log returns for the Apple stock and the algorithmic trading strategy based on SMAs need to be added up and the exponential function applied to arrive at the performance values:

```
In [14]: data['Returns'] = np.log(data[symbol] / data[symbol].shift(1))  ❶

In [15]: data['Strategy'] = data['Position'].shift(1) * data['Returns']  ❷

In [16]: data.round(4).head()
Out[16]:              AAPL.O     SMA1     SMA2  Position  Returns  Strategy
         Date
         2010-12-31  46.0800  45.2810  37.1207         1      NaN       NaN
         2011-01-03  47.0814  45.3497  37.1862         1   0.0215    0.0215
         2011-01-04  47.3271  45.4126  37.2525         1   0.0052    0.0052
         2011-01-05  47.7142  45.4661  37.3223         1   0.0081    0.0081
         2011-01-06  47.6757  45.5226  37.3921         1  -0.0008   -0.0008

In [17]: data.dropna(inplace=True)

In [18]: np.exp(data[['Returns', 'Strategy']].sum())  ❸
Out[18]: Returns     4.017148
         Strategy    5.811299
         dtype: float64

In [19]: data[['Returns', 'Strategy']].std() * 252 ** 0.5  ❹
Out[19]: Returns     0.250571
         Strategy    0.250407
         dtype: float64
```

❶ Calculates the log returns of the Apple stock (i.e., the benchmark investment).

❷ Multiplies the position values, shifted by one day, by the log returns of the Apple stock; the shift is required to avoid a foresight bias.[5]

❸ Sums up the log returns for the strategy and the benchmark investment and calculates the exponential value to arrive at the absolute performance.

❹ Calculates the annualized volatility for the strategy and the benchmark investment.

The numbers show that the algorithmic trading strategy indeed outperforms the benchmark investment of passively holding the Apple stock. Due to the type and characteristics of the strategy, the annualized volatility is the same, such that it

also outperforms the benchmark investment on a risk-adjusted basis.

To gain a better picture of the overall performance, Figure 15-3 shows the performance of the Apple stock and the algorithmic trading strategy over time:

```
In [20]: ax = data[['Returns', 'Strategy']].cumsum(
                    ).apply(np.exp).plot(figsize=(10, 6))
         data['Position'].plot(ax=ax, secondary_y='Position', style='--')
         ax.get_legend().set_bbox_to_anchor((0.25, 0.85));
```

Figure 15-3. Performance of Apple stock and SMA-based trading strategy over time

> **SIMPLIFICATIONS**
>
> The vectorized backtesting approach as introduced in this subsection is based on a number of simplifying assumptions. Among others, transactions costs (fixed fees, bid-ask spreads, lending costs, etc.) are not included. This might be justifiable for a trading strategy that leads to a few trades only over multiple years. It is also assumed that all trades take place at the end-of-day closing prices for the Apple stock. A more realistic backtesting approach would take these and other (market microstructure) elements into account.

Optimization

A natural question that arises is if the chosen parameters `SMA1=42` and `SMA2=252`

are the “right” ones. In general, investors prefer higher returns to lower returns *ceteris paribus*. Therefore, one might be inclined to search for those parameters that maximize the return over the relevant period. To this end, a brute force approach can be used that simply repeats the whole vectorized backtesting procedure for different parameter combinations, records the results, and does a ranking afterward. This is what the following code does:

```
In [21]: from itertools import product

In [22]: sma1 = range(20, 61, 4)  ❶
         sma2 = range(180, 281, 10)  ❷

In [23]: results = pd.DataFrame()
         for SMA1, SMA2 in product(sma1, sma2):  ❸
             data = pd.DataFrame(raw[symbol])
             data.dropna(inplace=True)
             data['Returns'] = np.log(data[symbol] / data[symbol].shift(1))
             data['SMA1'] = data[symbol].rolling(SMA1).mean()
             data['SMA2'] = data[symbol].rolling(SMA2).mean()
             data.dropna(inplace=True)
             data['Position'] = np.where(data['SMA1'] > data['SMA2'], 1, -1)
             data['Strategy'] = data['Position'].shift(1) * data['Returns']
             data.dropna(inplace=True)
             perf = np.exp(data[['Returns', 'Strategy']].sum())
             results = results.append(pd.DataFrame(
                         {'SMA1': SMA1, 'SMA2': SMA2,
                          'MARKET': perf['Returns'],
                          'STRATEGY': perf['Strategy'],
                          'OUT': perf['Strategy'] - perf['Returns']},
                          index=[0]), ignore_index=True)  ❹
```

❶ Specifies the parameter values for `SMA1`.

❷ Specifies the parameter values for `SMA2`.

❸ Combines all values for `SMA1` with those for `SMA2`.

❹ Records the vectorized backtesting results in a `DataFrame` object.

The following code gives an overview of the results and shows the seven best-performing parameter combinations of all those backtested. The ranking is implemented according to the outperformance of the algorithmic trading strategy compared to the benchmark investment. The performance of the benchmark investment varies since the choice of the `SMA2` parameter influences the length of the time interval and data set on which the vectorized backtest is implemented:

```
In [24]: results.info()
         <class 'pandas.core.frame.DataFrame'>
         RangeIndex: 121 entries, 0 to 120
         Data columns (total 5 columns):
         SMA1        121 non-null int64
         SMA2        121 non-null int64
         MARKET      121 non-null float64
         STRATEGY    121 non-null float64
         OUT         121 non-null float64
         dtypes: float64(3), int64(2)
         memory usage: 4.8 KB

In [25]: results.sort_values('OUT', ascending=False).head(7)
Out[25]:      SMA1  SMA2    MARKET  STRATEGY       OUT
         56     40   190  4.650342  7.175173  2.524831
         39     32   240  4.045619  6.558690  2.513071
         59     40   220  4.220272  6.544266  2.323994
         46     36   200  4.074753  6.389627  2.314874
         55     40   180  4.574979  6.857989  2.283010
         70     44   220  4.220272  6.469843  2.249571
         101    56   200  4.074753  6.319524  2.244772
```

According to the brute force–based optimization, `SMA1=40` and `SMA2=190` are the optimal parameters, leading to an outperformance of some 230 percentage points. However, this result is heavily dependent on the data set used and is prone to overfitting. A more rigorous approach would be to implement the optimization on one data set, the in-sample or training data set, and test it on another one, the out-of-sample or testing data set.

OVERFITTING

In general, any type of optimization, fitting, or training in the context of algorithmic trading strategies is prone to what is called *overfitting*. This means that parameters might be chosen that perform (exceptionally) well for the used data set but might perform (exceptionally) badly on other data sets or in practice.

Random Walk Hypothesis

The previous section introduces vectorized backtesting as an efficient tool to backtest algorithmic trading strategies. The single strategy backtested based on a single financial time series, namely historical end-of-day prices for the Apple

stock, outperforms the benchmark investment of simply going long on the Apple stock over the same period.

Although rather specific in nature, these results are in contrast to what the *random walk hypothesis* (RWH) predicts, namely that such predictive approaches should not yield any outperformance at all. The RWH postulates that prices in financial markets follow a random walk, or, in continuous time, an arithmetic Brownian motion without drift. The expected value of an arithmetic Brownian motion without drift at any point in the future equals its value today.[6] As a consequence, the best predictor for tomorrow's price, in a least-squares sense, is today's price if the RWH applies.

The consequences are summarized in the following quote:

> *For many years, economists, statisticians, and teachers of finance have been interested in developing and testing models of stock price behavior. One important model that has evolved from this research is the theory of random walks. This theory casts serious doubt on many other methods for describing and predicting stock price behavior—methods that have considerable popularity outside the academic world. For example, we shall see later that, if the random-walk theory is an accurate description of reality, then the various "technical" or "chartist" procedures for predicting stock prices are completely without value.*
>
> —Eugene F. Fama (1965)

The RWH is consistent with the *efficient markets hypothesis* (EMH), which, non-technically speaking, states that market prices reflect "all available information." Different degrees of efficiency are generally distinguished, such as *weak*, *semi-strong*, and *strong*, defining more specifically what "all available information" entails. Formally, such a definition can be based on the concept of an information set in theory and on a data set for programming purposes, as the following quote illustrates:

> *A market is efficient with respect to an information set S if it is impossible to make economic profits by trading on the basis of information set S.*
>
> —Michael Jensen (1978)

Using Python, the RWH can be tested for a specific case as follows. A financial time series of historical market prices is used for which a number of *lagged*

versions are created—say, five. OLS regression is then used to predict the market prices based on the lagged market prices created before. The basic idea is that the market prices from yesterday and four more days back can be used to predict today's market price.

The following Python code implements this idea and creates five lagged versions of the historical end-of-day closing levels of the S&P 500 stock index:

```
In [26]: symbol = '.SPX'

In [27]: data = pd.DataFrame(raw[symbol])

In [28]: lags = 5
         cols = []
         for lag in range(1, lags + 1):
             col = 'lag_{}'.format(lag)  ❶
             data[col] = data[symbol].shift(lag)  ❷
             cols.append(col)  ❸

In [29]: data.head(7)
Out[29]:                .SPX    lag_1    lag_2    lag_3    lag_4    lag_5
         Date
         2010-01-01      NaN      NaN      NaN      NaN      NaN      NaN
         2010-01-04  1132.99      NaN      NaN      NaN      NaN      NaN
         2010-01-05  1136.52  1132.99      NaN      NaN      NaN      NaN
         2010-01-06  1137.14  1136.52  1132.99      NaN      NaN      NaN
         2010-01-07  1141.69  1137.14  1136.52  1132.99      NaN      NaN
         2010-01-08  1144.98  1141.69  1137.14  1136.52  1132.99      NaN
         2010-01-11  1146.98  1144.98  1141.69  1137.14  1136.52  1132.99

In [30]: data.dropna(inplace=True)
```

❶ Defines a column name for the current `lag` value.

❷ Creates the lagged version of the market prices for the current `lag` value.

❸ Collects the column names for later reference.

Using `NumPy`, the OLS regression is straightforward to implement. As the optimal regression parameters show, `lag_1` indeed is the most important one in predicting the market price based on OLS regression. Its value is close to 1. The other four values are rather close to 0. Figure 15-4 visualizes the optimal regression parameter values.

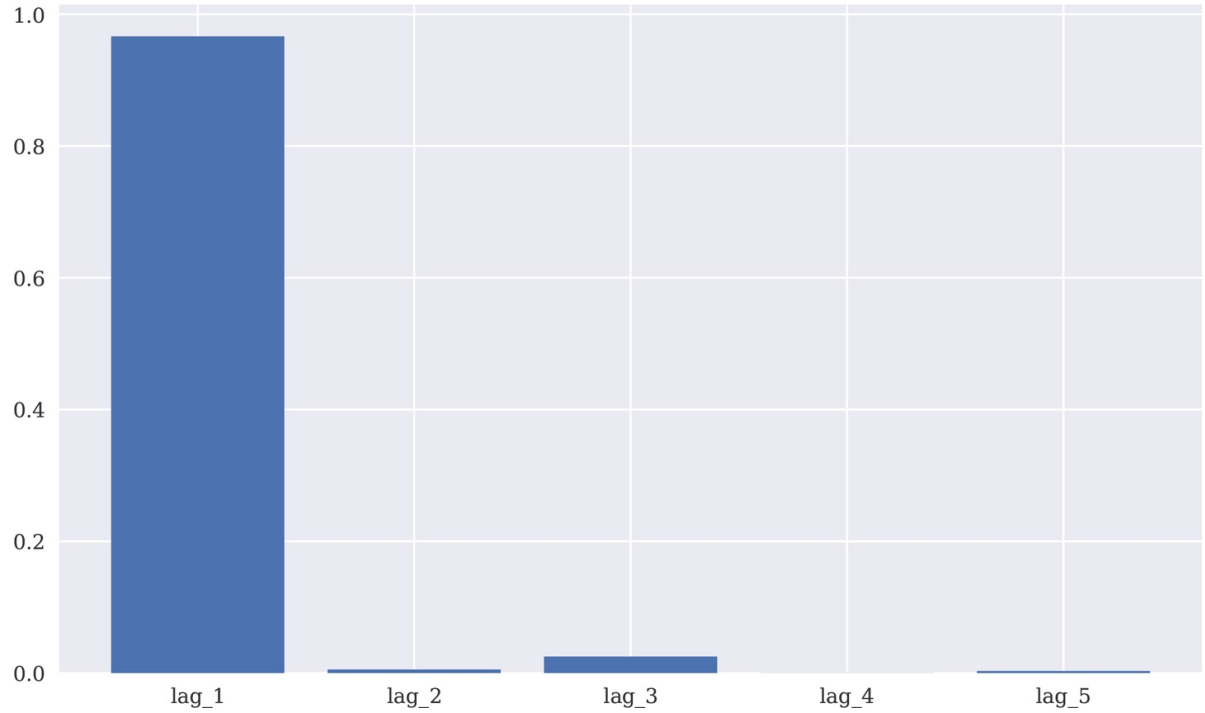

Figure 15-4. Optimal regression parameters from OLS regression for price prediction

When using the optimal results to visualize the prediction values as compared to the original index values for the S&P 500, it becomes obvious from Figure 15-5 that indeed `lag_1` is basically what is used to come up with the prediction value. Graphically speaking, the prediction line in Figure 15-5 is the original time series shifted by one day to the right (with some minor adjustments).

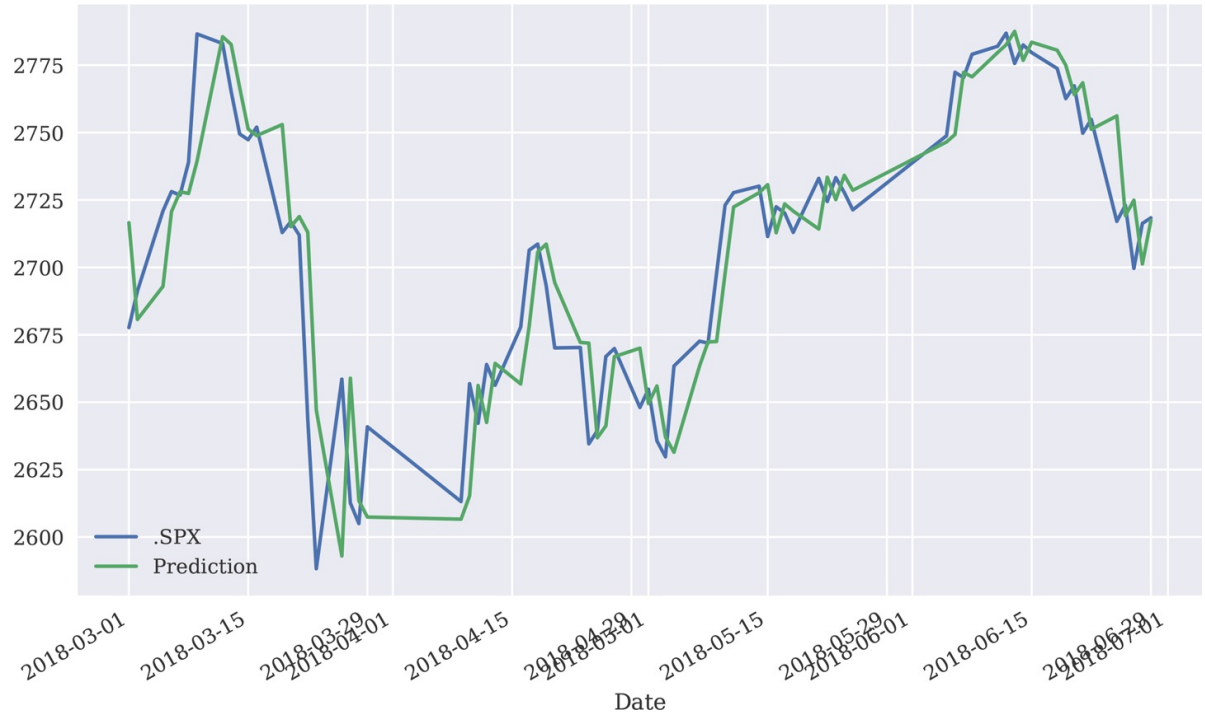

Figure 15-5. S&P 500 levels compared to prediction values from OLS regression

All in all, the brief analysis in this section reveals some support for both the RWH and the EMH. For sure, the analysis is done for a single stock index only and uses a rather specific parameterization—but this can easily be widened to incorporate multiple financial instruments across multiple asset classes, different values for the number of lags, etc. In general, one will find out that the results are qualitatively more or less the same. After all, the RWH and EMH are among the financial theories that have broad empirical support. In that sense, any algorithmic trading strategy must prove its worth by proving that the RWH does not apply in general. This for sure is a tough hurdle.

Linear OLS Regression

This section applies *linear OLS regression* to predict the direction of market movements based on historical log returns. To keep things simple, only two features are used. The first feature (`lag_1`) represents the log returns of the financial time series lagged by *one day*. The second feature (`lag_2`) lags the log returns by *two days*. Log returns—in contrast to prices—are *stationary* in general, which often is a necessary condition for the application of statistical and ML algorithms.

The basic idea behind the usage of lagged log returns as features is that they might be informative in predicting future returns. For example, one might hypothesize that after two downward movements an upward movement is more likely ("mean reversion"), or, to the contrary, that another downward movement is more likely ("momentum" or "trend"). The application of regression techniques allows the formalization of such informal reasonings.

The Data

First, the importing and preparation of the data set. Figure 15-6 shows the frequency distribution of the daily historical log returns for the EUR/USD exchange rate. They are the basis for the features as well as the labels to be used in what follows:

```
In [3]: raw = pd.read_csv('../../source/tr_eikon_eod_data.csv',
                          index_col=0, parse_dates=True).dropna()

In [4]: raw.columns
Out[4]: Index(['AAPL.O', 'MSFT.O', 'INTC.O', 'AMZN.O', 'GS.N', 'SPY', '.SPX',
               '.VIX', 'EUR=', 'XAU=', 'GDX', 'GLD'],
              dtype='object')

In [5]: symbol = 'EUR='

In [6]: data = pd.DataFrame(raw[symbol])

In [7]: data['returns'] = np.log(data / data.shift(1))

In [8]: data.dropna(inplace=True)

In [9]: data['direction'] = np.sign(data['returns']).astype(int)

In [10]: data.head()
Out[10]:              EUR=   returns  direction
         Date
         2010-01-05  1.4368 -0.002988         -1
         2010-01-06  1.4412  0.003058          1
         2010-01-07  1.4318 -0.006544         -1
         2010-01-08  1.4412  0.006544          1
         2010-01-11  1.4513  0.006984          1

In [11]: data['returns'].hist(bins=35, figsize=(10, 6));
```

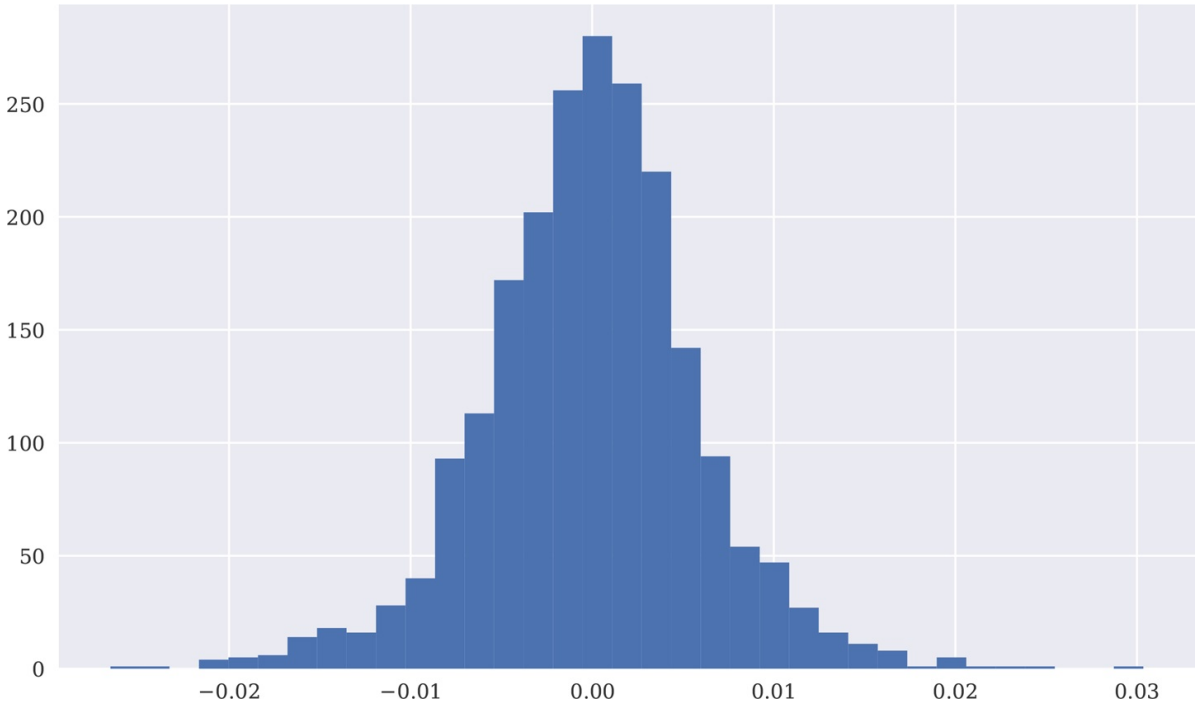

Figure 15-6. Histogram of log returns for EUR/USD exchange rate

Second, the code that creates the features data by lagging the log returns and visualizes it in combination with the returns data (see Figure 15-7):

```
In [12]: lags = 2

In [13]: def create_lags(data):
             global cols
             cols = []
             for lag in range(1, lags + 1):
                 col = 'lag_{}'.format(lag)
                 data[col] = data['returns'].shift(lag)
                 cols.append(col)

In [14]: create_lags(data)

In [15]: data.head()
Out[15]:              EUR=   returns  direction     lag_1     lag_2
         Date
         2010-01-05  1.4368 -0.002988         -1       NaN       NaN
         2010-01-06  1.4412  0.003058          1 -0.002988       NaN
         2010-01-07  1.4318 -0.006544         -1  0.003058 -0.002988
         2010-01-08  1.4412  0.006544          1 -0.006544  0.003058
         2010-01-11  1.4513  0.006984          1  0.006544 -0.006544

In [16]: data.dropna(inplace=True)
```

```
In [17]: data.plot.scatter(x='lag_1', y='lag_2', c='returns',
                           cmap='coolwarm', figsize=(10, 6), colorbar=True)
         plt.axvline(0, c='r', ls='--')
         plt.axhline(0, c='r', ls='--');
```

Figure 15-7. Scatter plot based on features and labels data

Regression

With the data set completed, linear OLS regression can be applied to learn about any potential (linear) relationships, to predict market movement based on the features, and to backtest a trading strategy based on the predictions. Two basic approaches are available: using the *log returns* or only the *direction data* as the dependent variable during the regression. In any case, predictions are real-valued and therefore transformed to either `+1` or `-1` to only work with the direction of the prediction:

```
In [18]: from sklearn.linear_model import LinearRegression  ❶

In [19]: model = LinearRegression()  ❶

In [20]: data['pos_ols_1'] = model.fit(data[cols],
                                       data['returns']).predict(data[cols])  ❷

In [21]: data['pos_ols_2'] = model.fit(data[cols],
                                       data['direction']).predict(data[cols])  ❸
```

```
In [22]: data[['pos_ols_1', 'pos_ols_2']].head()
Out[22]:             pos_ols_1  pos_ols_2
         Date
         2010-01-07  -0.000166  -0.000086
         2010-01-08   0.000017   0.040404
         2010-01-11  -0.000244  -0.011756
         2010-01-12  -0.000139  -0.043398
         2010-01-13  -0.000022   0.002237

In [23]: data[['pos_ols_1', 'pos_ols_2']] = np.where(
                     data[['pos_ols_1', 'pos_ols_2']] > 0, 1, -1)  ❹

In [24]: data['pos_ols_1'].value_counts()  ❺
Out[24]: -1    1847
          1     288
         Name: pos_ols_1, dtype: int64

In [25]: data['pos_ols_2'].value_counts()  ❺
Out[25]:  1    1377
         -1     758
         Name: pos_ols_2, dtype: int64

In [26]: (data['pos_ols_1'].diff() != 0).sum()  ❻
Out[26]: 555

In [27]: (data['pos_ols_2'].diff() != 0).sum()  ❻
Out[27]: 762
```

❶ The linear OLS regression implementation from `scikit-learn` is used.
❷ The regression is implemented on the *log returns* directly …
❸ … and on the *direction data* which is of primary interest.
❹ The real-valued predictions are transformed to directional values (`+1`, `-1`).
❺ The two approaches yield different directional predictions in general.
❻ However, both lead to a relatively large number of trades over time.

Equipped with the directional prediction, vectorized backtesting can be applied to judge the performance of the resulting trading strategies. At this stage, the analysis is based on a number of simplifying assumptions, such as “zero transaction costs” and the usage of the same data set for both training and testing. Under these assumptions, however, both regression-based strategies outperform the benchmark passive investment, while only the strategy trained on the direction of the market shows a positive overall performance (Figure 15-8):

```
In [28]: data['strat_ols_1'] = data['pos_ols_1'] * data['returns']

In [29]: data['strat_ols_2'] = data['pos_ols_2'] * data['returns']

In [30]: data[['returns', 'strat_ols_1', 'strat_ols_2']].sum().apply(np.exp)
Out[30]: returns        0.810644
         strat_ols_1    0.942422
         strat_ols_2    1.339286
         dtype: float64

In [31]: (data['direction'] == data['pos_ols_1']).value_counts()  ❶
Out[31]: False    1093
         True     1042
         dtype: int64

In [32]: (data['direction'] == data['pos_ols_2']).value_counts()  ❶
Out[32]: True     1096
         False    1039
         dtype: int64

In [33]: data[['returns', 'strat_ols_1', 'strat_ols_2']].cumsum(
                ).apply(np.exp).plot(figsize=(10, 6));
```

❶ Shows the number of correct and false predictions by the strategies.

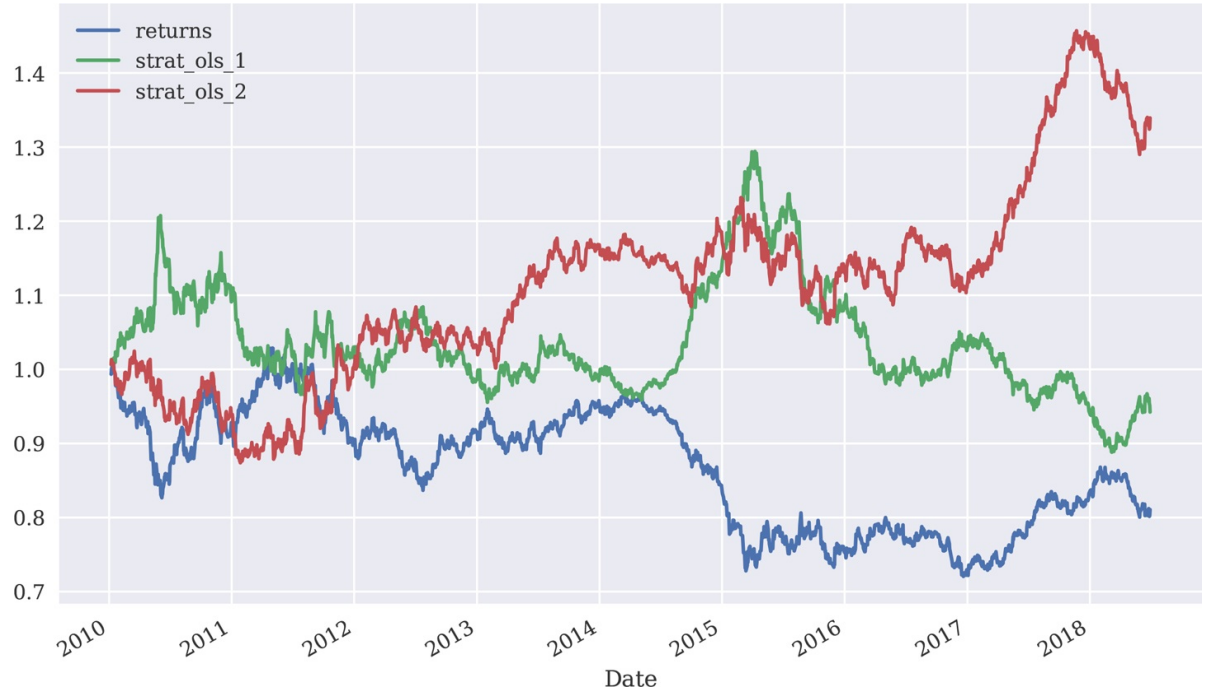

Figure 15-8. Performance of EUR/USD and regression-based strategies over time

Clustering

This section applies *k*-means clustering, as introduced in "Machine Learning", to financial time series data to automatically come up with clusters that are used to formulate a trading strategy. The idea is that the algorithm identifies two clusters of feature values that predict either an upward movement or a downward movement.

The following code applies the *k*-means algorithm to the two features as used before. Figure 15-9 visualizes the two clusters:

```
In [34]: from sklearn.cluster import KMeans

In [35]: model = KMeans(n_clusters=2, random_state=0)  ❶

In [36]: model.fit(data[cols])
Out[36]: KMeans(algorithm='auto', copy_x=True, init='k-means++', max_iter=300,
             n_clusters=2, n_init=10, n_jobs=None, precompute_distances='auto',
             random_state=0, tol=0.0001, verbose=0)

In [37]: data['pos_clus'] = model.predict(data[cols])

In [38]: data['pos_clus'] = np.where(data['pos_clus'] == 1, -1, 1)  ❷

In [39]: data['pos_clus'].values
Out[39]: array([-1,  1, -1, ...,  1,  1, -1])

In [40]: plt.figure(figsize=(10, 6))
         plt.scatter(data[cols].iloc[:, 0], data[cols].iloc[:, 1],
                     c=data['pos_clus'], cmap='coolwarm');
```

❶ Two clusters are chosen for the algorithm.

❷ Given the cluster values, the position is chosen.

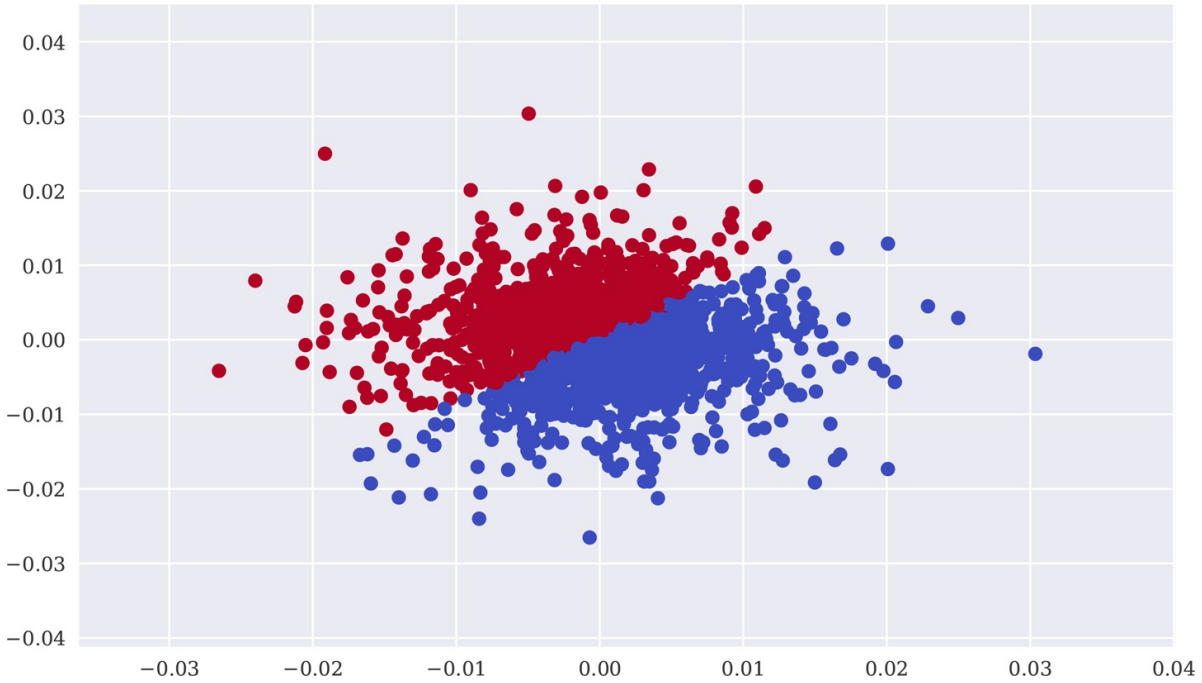

Figure 15-9. Two clusters as identified by the k-means algorithm

Admittedly, this approach is quite arbitrary in this context—after all, how should the algorithm know what one is looking for? However, the resulting trading strategy shows a slight outperformance at the end compared to the benchmark passive investment (see Figure 15-10). It is noteworthy that no guidance (supervision) is given and that the *hit ratio*—i.e., the number of correct predictions in relationship to all predictions made—is less than 50%:

```
In [41]: data['strat_clus'] = data['pos_clus'] * data['returns']

In [42]: data[['returns', 'strat_clus']].sum().apply(np.exp)
Out[42]: returns       0.810644
         strat_clus    1.277133
         dtype: float64

In [43]: (data['direction'] == data['pos_clus']).value_counts()
Out[43]: True     1077
         False    1058
         dtype: int64

In [44]: data[['returns', 'strat_clus']].cumsum(
                ).apply(np.exp).plot(figsize=(10, 6));
```

Figure 15-10. Performance of EUR/USD and k-means-based strategy over time

Frequency Approach

Beyond more sophisticated algorithms and techniques, one might come up with the idea of just implementing a *frequency approach* to predict directional movements in financial markets. To this end, one might transform the two real-valued features to binary ones and assess the probability of an upward and a downward movement, respectively, from the historical observations of such movements, given the four possible combinations for the two binary features ((0, 0), (0, 1), (1, 0), (1, 1)).

Making use of the data analysis capabilities of `pandas`, such an approach is relatively easy to implement:

```
In [45]: def create_bins(data, bins=[0]):
             global cols_bin
             cols_bin = []
             for col in cols:
                 col_bin = col + '_bin'
                 data[col_bin] = np.digitize(data[col], bins=bins)  ❶
                 cols_bin.append(col_bin)

In [46]: create_bins(data)
```

```
In [52]: data['pos_freq'] = np.where(data[cols_bin].sum(axis=1) == 2, -1, 1)  ❶

In [53]: (data['direction'] == data['pos_freq']).value_counts()
Out[53]: True     1102
         False    1033
         dtype: int64

In [54]: data['strat_freq'] = data['pos_freq'] * data['returns']

In [55]: data[['returns', 'strat_freq']].sum().apply(np.exp)
Out[55]: returns       0.810644
         strat_freq    0.989513
         dtype: float64

In [56]: data[['returns', 'strat_freq']].cumsum(
                ).apply(np.exp).plot(figsize=(10, 6));
```

❶ Translates the findings given the frequencies to a trading strategy.

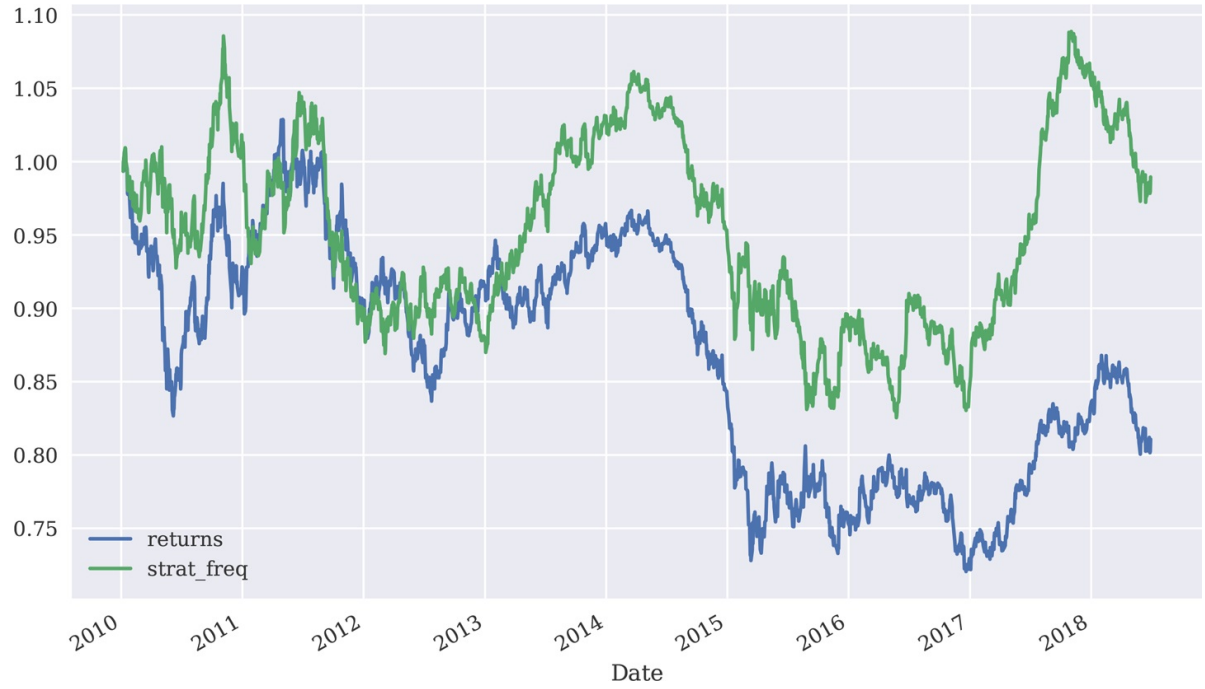

Figure 15-11. Performance of EUR/USD and frequency-based trading strategy over time

Classification

This section applies the classification algorithms from ML (as introduced in "Machine Learning") to the problem of predicting the direction of price movements in financial markets. With that background and the examples from

previous sections, the application of the logistic regression, Gaussian Naive Bayes, and support vector machine approaches is as straightforward as applying them to smaller sample data sets.

Two Binary Features

First, a fitting of the models based on the binary feature values and the derivation of the resulting position values:

```
In [57]: from sklearn import linear_model
         from sklearn.naive_bayes import GaussianNB
         from sklearn.svm import SVC

In [58]: C = 1

In [59]: models = {
             'log_reg': linear_model.LogisticRegression(C=C),
             'gauss_nb': GaussianNB(),
             'svm': SVC(C=C)
         }

In [60]: def fit_models(data):  ❶
             mfit = {model: models[model].fit(data[cols_bin],
                                              data['direction'])
                     for model in models.keys()}

In [61]: fit_models(data)

In [62]: def derive_positions(data):  ❷
             for model in models.keys():
                 data['pos_' + model] = models[model].predict(data[cols_bin])

In [63]: derive_positions(data)
```

❶ A function that fits all models.

❷ A function that derives all position values from the fitted models.

Second, the vectorized backtesting of the resulting trading strategies. Figure 15-12 visualizes the performance over time:

```
In [64]: def evaluate_strats(data):  ❶
             global sel
             sel = []
             for model in models.keys():
                 col = 'strat_' + model
```

```
                data[col] = data['pos_' + model] * data['returns']
                sel.append(col)
            sel.insert(0, 'returns')

In [65]: evaluate_strats(data)

In [66]: sel.insert(1, 'strat_freq')

In [67]: data[sel].sum().apply(np.exp)  ❷
Out[67]: returns           0.810644
         strat_freq        0.989513
         strat_log_reg     1.243322
         strat_gauss_nb    1.243322
         strat_svm         0.989513
         dtype: float64

In [68]: data[sel].cumsum().apply(np.exp).plot(figsize=(10, 6));
```

❶ A function that evaluates all resulting trading strategies.

❷ Some strategies might show the exact same performance.

Figure 15-12. Performance of EUR/USD and classification-based trading strategies (two binary lags) over time

Five Binary Features

In an attempt to improve the strategies' performance, the following code works

with five binary lags instead of two. In particular, the performance of the SVM-based strategy is significantly improved (see Figure 15-13). On the other hand, the performance of the LR- and GNB-based strategies is worse:

```
In [69]: data = pd.DataFrame(raw[symbol])

In [70]: data['returns'] = np.log(data / data.shift(1))

In [71]: data['direction'] = np.sign(data['returns'])

In [72]: lags = 5  ❶
         create_lags(data)
         data.dropna(inplace=True)

In [73]: create_bins(data)  ❷
         cols_bin
Out[73]: ['lag_1_bin', 'lag_2_bin', 'lag_3_bin', 'lag_4_bin', 'lag_5_bin']

In [74]: data[cols_bin].head()
Out[74]:             lag_1_bin  lag_2_bin  lag_3_bin  lag_4_bin  lag_5_bin
         Date
         2010-01-12          1          1          0          1          0
         2010-01-13          0          1          1          0          1
         2010-01-14          1          0          1          1          0
         2010-01-15          0          1          0          1          1
         2010-01-19          0          0          1          0          1

In [75]: data.dropna(inplace=True)

In [76]: fit_models(data)

In [77]: derive_positions(data)

In [78]: evaluate_strats(data)

In [79]: data[sel].sum().apply(np.exp)
Out[79]: returns           0.805002
         strat_log_reg     0.971623
         strat_gauss_nb    0.986420
         strat_svm         1.452406
         dtype: float64

In [80]: data[sel].cumsum().apply(np.exp).plot(figsize=(10, 6));
```

❶ Five lags of the log returns series are now used.

❷ The real-valued features data is transformed to binary data.

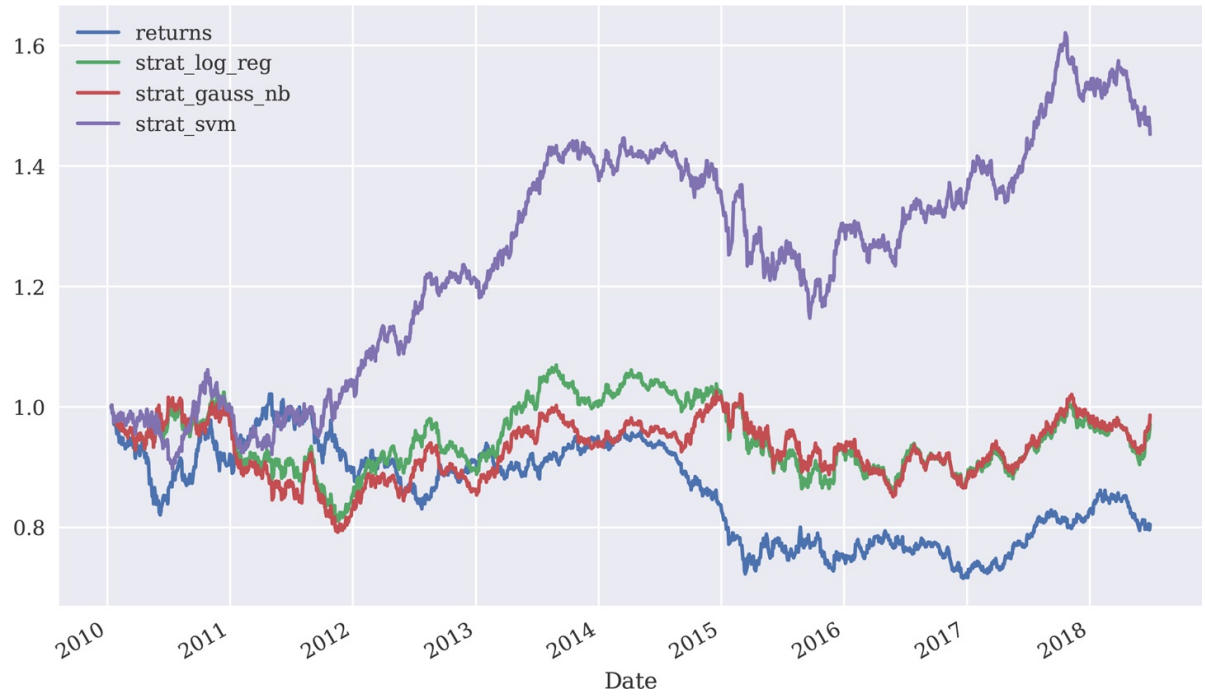

Figure 15-13. Performance of EUR/USD and classification-based trading strategies (five binary lags) over time

Five Digitized Features

Finally, the following code uses the first and second moment of the historical log returns to digitize the features data, allowing for more possible feature value combinations. This improves the performance of all classification algorithms used, but for SVM the improvement is again most pronounced (see Figure 15-14):

```
In [81]: mu = data['returns'].mean()  ❶
         v = data['returns'].std()  ❷

In [82]: bins = [mu - v, mu, mu + v]  ❸
         bins  ❸
Out[82]: [-0.006033537040418665, -0.00010174015279231306, 0.005830056734834039]

In [83]: create_bins(data, bins)

In [84]: data[cols_bin].head()
Out[84]:             lag_1_bin  lag_2_bin  lag_3_bin  lag_4_bin  lag_5_bin
         Date
         2010-01-12          3          3          0          2          1
         2010-01-13          1          3          3          0          2
         2010-01-14          2          1          3          3          0
```

```
2010-01-15          1          2          1          3          3
2010-01-19          0          1          2          1          3

In [85]: fit_models(data)

In [86]: derive_positions(data)

In [87]: evaluate_strats(data)

In [88]: data[sel].sum().apply(np.exp)
Out[88]: returns           0.805002
         strat_log_reg     1.431120
         strat_gauss_nb    1.815304
         strat_svm         5.653433
         dtype: float64

In [89]: data[sel].cumsum().apply(np.exp).plot(figsize=(10, 6));
```

❶ The mean log return and …

❷ … the standard deviation are used …

❸ … to digitize the features data.

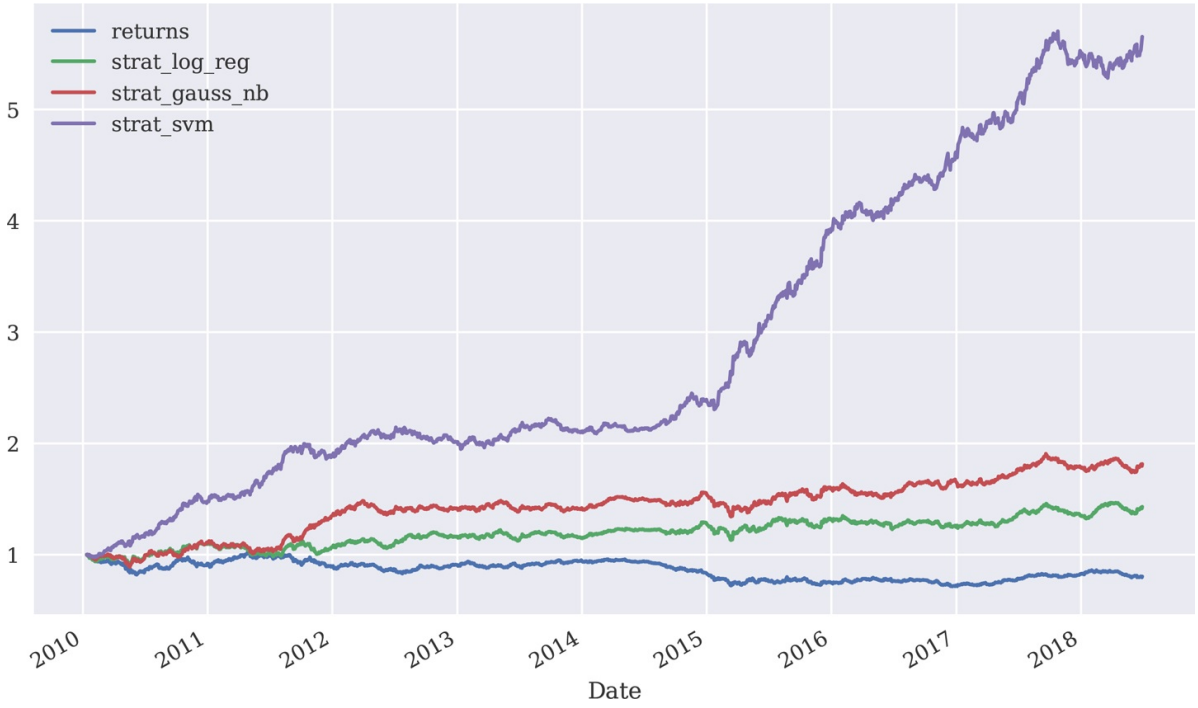

Figure 15-14. Performance of EUR/USD and classification-based trading strategies (five digitized lags) over time

TYPES OF FEATURES

This chapter exclusively works with lagged return data as features data, mostly in binarized or digitized form. This is mainly done for convenience, since such features data can be derived from the financial time series itself. However, in practical applications the features data can be gained from a wealth of different data sources and might include other financial time series and statistics derived thereof, macroeconomic data, company financial indicators, or news articles. Refer to López de Prado (2018) for an in-depth discussion of this topic. There are also Python packages for automated time series feature extraction available, such as `tsfresh`.

Sequential Train-Test Split

To better judge the performance of the classification algorithms, the code that follows implements a *sequential* train-test split. The idea here is to simulate the situation where only data up to a certain point in time is available on which to train an ML algorithm. During live trading, the algorithm is then faced with data it has never seen before. This is where the algorithm must prove its worth. In this particular case, all classification algorithms outperform—under the simplified assumptions from before—the passive benchmark investment, but only the GNB and LR algorithms achieve a positive absolute performance (Figure 15-15):

```
In [90]: split = int(len(data) * 0.5)

In [91]: train = data.iloc[:split].copy()  ❶

In [92]: fit_models(train)  ❶

In [93]: test = data.iloc[split:].copy()  ❷

In [94]: derive_positions(test)  ❷

In [95]: evaluate_strats(test)  ❷

In [96]: test[sel].sum().apply(np.exp)
Out[96]: returns           0.850291
         strat_log_reg     0.962989
         strat_gauss_nb    0.941172
         strat_svm         1.048966
         dtype: float64

In [97]: test[sel].cumsum().apply(np.exp).plot(figsize=(10, 6));
```

❶ Trains all classification algorithms on the training data.

❷ Tests all classification algorithms on the test data.

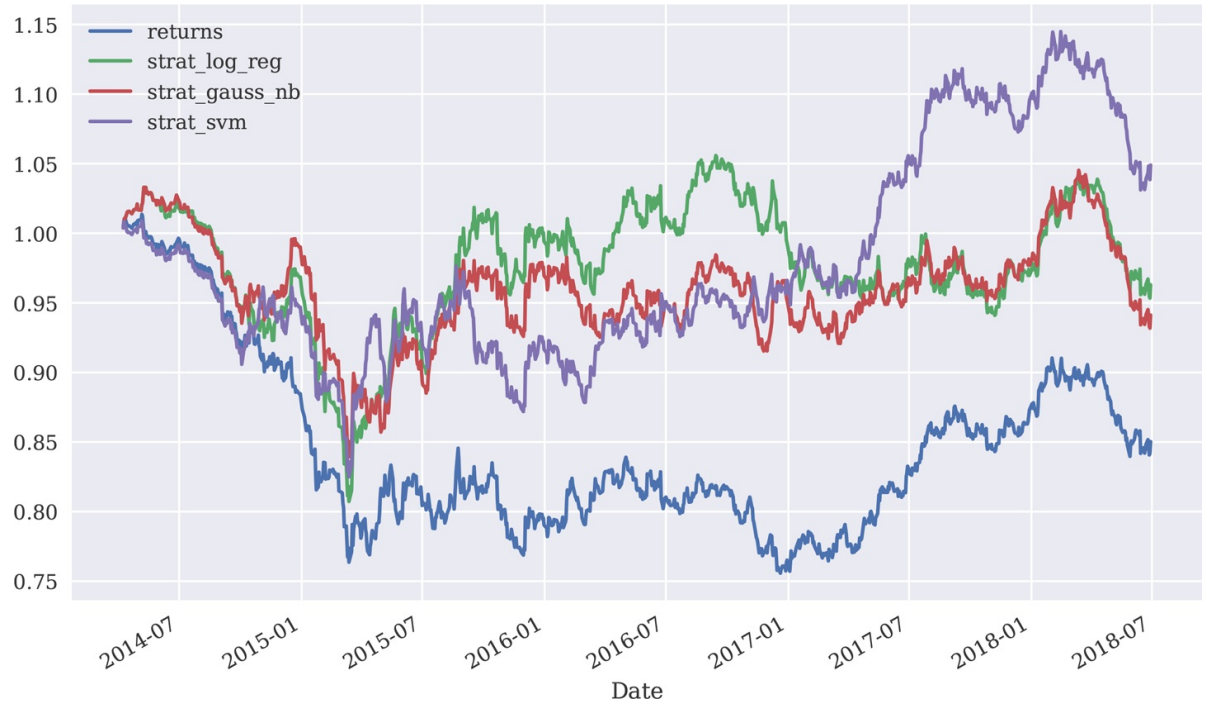

Figure 15-15. Performance of EUR/USD and classification-based trading strategies (sequential train-test split)

Randomized Train-Test Split

The classification algorithms are trained and tested on binary or digitized features data. The idea is that the feature value patterns allow a prediction of future market movements with a better hit ratio than 50%. Implicitly, it is assumed that the patterns' predictive power persists over time. In that sense, it shouldn't make (too much of) a difference on which part of the data an algorithm is trained and on which part of the data it is tested—implying that one can break up the temporal sequence of the data for training and testing.

A typical way to do this is a *randomized* train-test split to test the performance of the classification algorithms out-of-sample—again trying to emulate reality, where an algorithm during trading is faced with new data on a continuous basis. The approach used is the same as that applied to the sample data in "Train-test splits: Support vector machines". Based on this approach, the SVM algorithm shows again the best performance out-of-sample (see Figure 15-16):

```
In [98]: from sklearn.model_selection import train_test_split

In [99]: train, test = train_test_split(data, test_size=0.5,
```

```
                                        shuffle=True, random_state=100)

In [100]: train = train.copy().sort_index()  ❶

In [101]: train[cols_bin].head()
Out[101]:             lag_1_bin  lag_2_bin  lag_3_bin  lag_4_bin  lag_5_bin
          Date
          2010-01-12          3          3          0          2          1
          2010-01-13          1          3          3          0          2
          2010-01-14          2          1          3          3          0
          2010-01-15          1          2          1          3          3
          2010-01-20          1          0          1          2          1

In [102]: test = test.copy().sort_index()  ❶

In [103]: fit_models(train)

In [104]: derive_positions(test)

In [105]: evaluate_strats(test)

In [106]: test[sel].sum().apply(np.exp)
Out[106]: returns           0.878078
          strat_log_reg     0.735893
          strat_gauss_nb    0.765009
          strat_svm         0.695428
          dtype: float64

In [107]: test[sel].cumsum().apply(np.exp).plot(figsize=(10, 6));
```

❶ Train and test data sets are copied and brought back in temporal order.

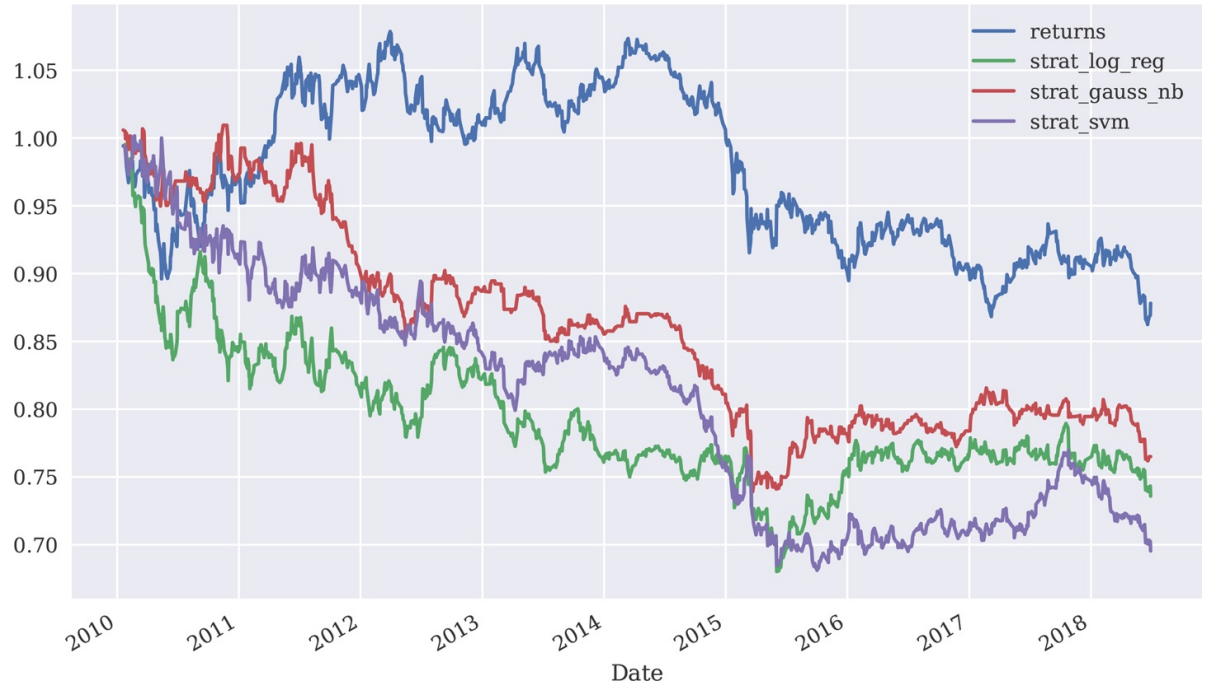

Figure 15-16. Performance of EUR/USD and classification-based trading strategies (randomized train-test split)

Deep Neural Networks

Deep neural networks (DNNs) try to emulate the functioning of the human brain. They are in general composed of an input layer (the features), an output layer (the labels), and a number of hidden layers. The presence of hidden layers is what makes a neural network *deep*. It allows it to learn more complex relationships and to perform better on a number of problem types. When applying DNNs one generally speaks of *deep learning* instead of machine learning. For an introduction to this field, refer to Géron (2017) or Gibson and Patterson (2017).

DNNs with scikit-learn

This section applies the `MLPClassifier` algorithm from `scikit-learn`, as introduced in "Deep neural networks". First, it is trained and tested on the whole data set, using the digitized features. The algorithm achieves exceptional performance in-sample (see Figure 15-17), which illustrates the power of DNNs for this type of problem. It also hints at strong overfitting, since the performance indeed seems unrealistically good:

```
In [108]: from sklearn.neural_network import MLPClassifier

In [109]: model = MLPClassifier(solver='lbfgs', alpha=1e-5,
                                hidden_layer_sizes=2 * [250],
                                random_state=1)

In [110]: %time model.fit(data[cols_bin], data['direction'])
          CPU times: user 16.1 s, sys: 156 ms, total: 16.2 s
          Wall time: 9.85 s

Out[110]: MLPClassifier(activation='relu', alpha=1e-05, batch_size='auto',
           beta_1=0.9,
                 beta_2=0.999, early_stopping=False, epsilon=1e-08,
                 hidden_layer_sizes=[250, 250], learning_rate='constant',
                 learning_rate_init=0.001, max_iter=200, momentum=0.9,
                 n_iter_no_change=10, nesterovs_momentum=True, power_t=0.5,
                 random_state=1, shuffle=True, solver='lbfgs', tol=0.0001,
                 validation_fraction=0.1, verbose=False, warm_start=False)

In [111]: data['pos_dnn_sk'] = model.predict(data[cols_bin])

In [112]: data['strat_dnn_sk'] = data['pos_dnn_sk'] * data['returns']

In [113]: data[['returns', 'strat_dnn_sk']].sum().apply(np.exp)
Out[113]: returns          0.805002
          strat_dnn_sk    35.156677
          dtype: float64

In [114]: data[['returns', 'strat_dnn_sk']].cumsum().apply(
                      np.exp).plot(figsize=(10, 6));
```

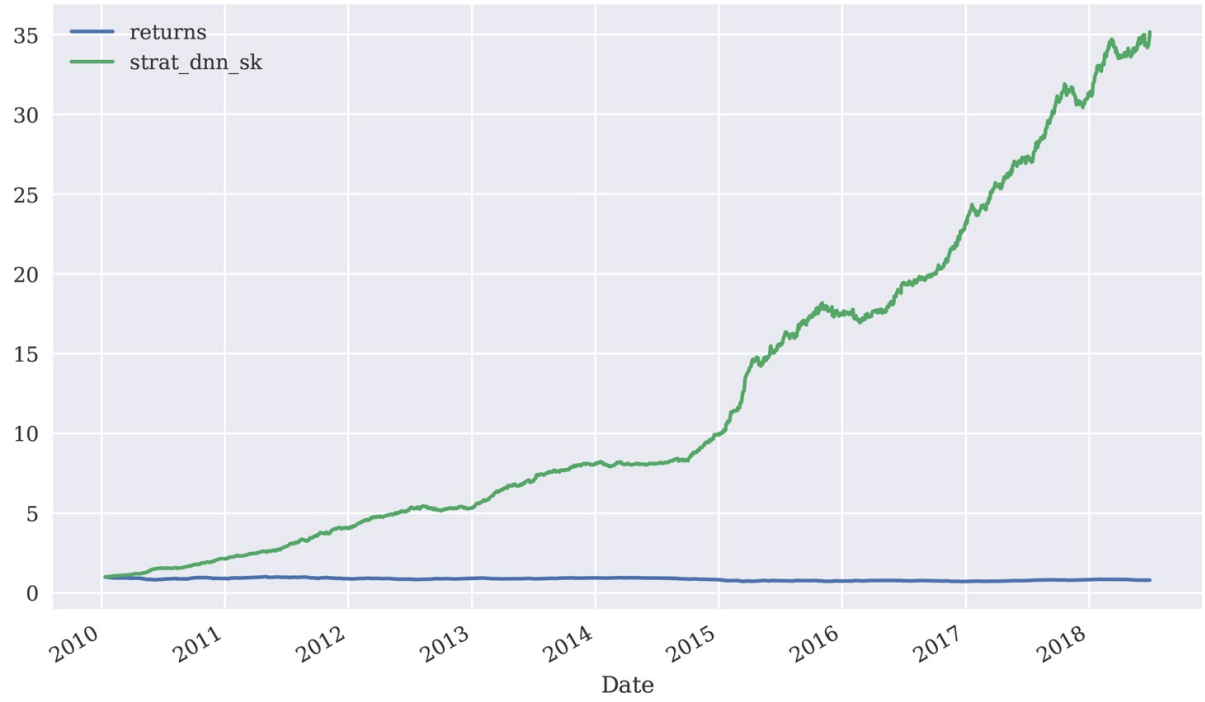

Figure 15-17. Performance of EUR/USD and DNN-based trading strategy (scikit-learn, in-sample)

To avoid overfitting of the DNN model, a randomized train-test split is applied next. The algorithm again outperforms the passive benchmark investment and achieves a positive absolute performance (Figure 15-18). However, the results seem more realistic now:

```
In [115]: train, test = train_test_split(data, test_size=0.5,
                                         random_state=100)

In [116]: train = train.copy().sort_index()

In [117]: test = test.copy().sort_index()

In [118]: model = MLPClassifier(solver='lbfgs', alpha=1e-5, max_iter=500,
                                hidden_layer_sizes=3 * [500], random_state=1)  ❶

In [119]: %time model.fit(train[cols_bin], train['direction'])
          CPU times: user 2min 26s, sys: 1.02 s, total: 2min 27s
          Wall time: 1min 31s

Out[119]: MLPClassifier(activation='relu', alpha=1e-05, batch_size='auto',
           beta_1=0.9,
                 beta_2=0.999, early_stopping=False, epsilon=1e-08,
                 hidden_layer_sizes=[500, 500, 500], learning_rate='constant',
                 learning_rate_init=0.001, max_iter=500, momentum=0.9,
                 n_iter_no_change=10, nesterovs_momentum=True, power_t=0.5,
```

```
                    random_state=1, shuffle=True, solver='lbfgs', tol=0.0001,
                    validation_fraction=0.1, verbose=False, warm_start=False)

In [120]: test['pos_dnn_sk'] = model.predict(test[cols_bin])

In [121]: test['strat_dnn_sk'] = test['pos_dnn_sk'] * test['returns']

In [122]: test[['returns', 'strat_dnn_sk']].sum().apply(np.exp)
Out[122]: returns         0.878078
          strat_dnn_sk    1.242042
          dtype: float64

In [123]: test[['returns', 'strat_dnn_sk']].cumsum(
                        ).apply(np.exp).plot(figsize=(10, 6));
```

❶ Increases the number of hidden layers and hidden units.

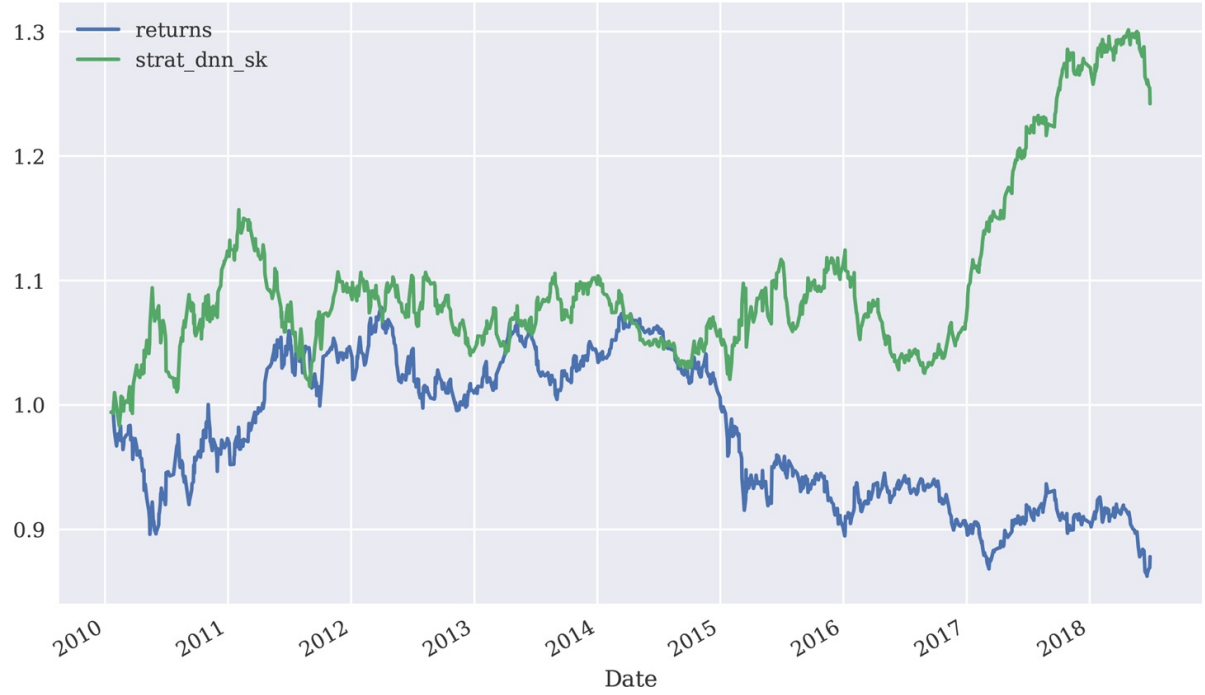

Figure 15-18. Performance of EUR/USD and DNN-based trading strategy (scikit-learn, randomized train-test split)

DNNs with TensorFlow

`TensorFlow` has become a popular package for deep learning. It is developed and supported by Google Inc. and applied there to a great variety of machine learning problems. Zedah and Ramsundar (2018) cover `TensorFlow` for deep learning in depth.

As with `scikit-learn`, the application of the `DNNClassifier` algorithm from `TensorFlow` to derive an algorithmic trading strategy is straightforward given the background from "Deep neural networks". The training and test data is the same as before. First, the training of the model. In-sample, the algorithm outperforms the passive benchmark investment and shows a considerable absolute return (see Figure 15-19), again hinting at overfitting:

```
In [124]: import tensorflow as tf
          tf.logging.set_verbosity(tf.logging.ERROR)

In [125]: fc = [tf.contrib.layers.real_valued_column('lags', dimension=lags)]

In [126]: model = tf.contrib.learn.DNNClassifier(hidden_units=3 * [500],
                                                 n_classes=len(bins) + 1,
                                                 feature_columns=fc)

In [127]: def input_fn():
              fc = {'lags': tf.constant(data[cols_bin].values)}
              la = tf.constant(data['direction'].apply(
                              lambda x: 0 if x < 0 else 1).values,
                              shape=[data['direction'].size, 1])
              return fc, la

In [128]: %time model.fit(input_fn=input_fn, steps=250)  ❶
          CPU times: user 2min 7s, sys: 8.85 s, total: 2min 16s
          Wall time: 49 s

Out[128]: DNNClassifier(params={'head':
           <tensorflow.contrib.learn.python.learn.estimators.head._MultiClassHead
           object at 0x1a19acf898>, 'hidden_units': [500, 500, 500],
           'feature_columns': (_RealValuedColumn(column_name='lags', dimension=5,
           default_value=None, dtype=tf.float32, normalizer=None),), 'optimizer':
           None, 'activation_fn': <function relu at 0x1161441e0>, 'dropout':
           None, 'gradient_clip_norm': None, 'embedding_lr_multipliers': None,
           'input_layer_min_slice_size': None})

In [129]: model.evaluate(input_fn=input_fn, steps=1)  ❷
Out[129]: {'loss': 0.6879357, 'accuracy': 0.5379925, 'global_step': 250}

In [130]: pred = np.array(list(model.predict(input_fn=input_fn)))  ❷
          pred[:10]  ❷
Out[130]: array([0, 0, 0, 0, 0, 1, 0, 1, 1, 0])

In [131]: data['pos_dnn_tf'] = np.where(pred > 0, 1, -1)  ❸

In [132]: data['strat_dnn_tf'] = data['pos_dnn_tf'] * data['returns']
```

```
In [133]: data[['returns', 'strat_dnn_tf']].sum().apply(np.exp)
Out[133]: returns         0.805002
          strat_dnn_tf    2.437222
          dtype: float64

In [134]: data[['returns', 'strat_dnn_tf']].cumsum(
                      ).apply(np.exp).plot(figsize=(10, 6));
```

❶ The time needed for training might be considerable.

❷ The binary predictions (`0`, `1`) …

❸ … need to be transformed to market positions (`-1`, `+1`).

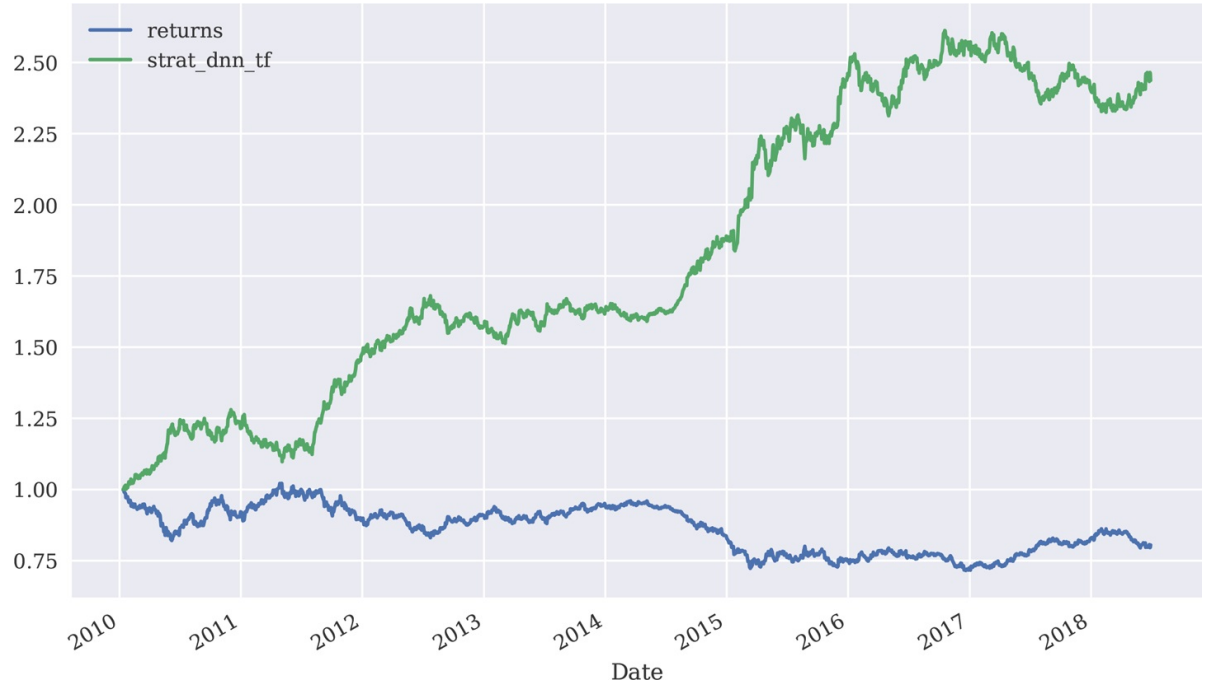

Figure 15-19. Performance of EUR/USD and DNN-based trading strategy (TensorFlow, in-sample)

The following code again implements a randomized train-test split to get a more realistic view of the performance of the DNN-based algorithmic trading strategy. The performance is, as expected, worse out-of-sample (see Figure 15-20). In addition, given the specific parameterization the `TensorFlow` `DNNClassifier` underperforms the `scikit-learn` `MLPClassifier` algorithm by quite few percentage points:

```
In [135]: model = tf.contrib.learn.DNNClassifier(hidden_units=3 * [500],
                                                  n_classes=len(bins) + 1,
                                                  feature_columns=fc)
```

```
In [136]: data = train

In [137]: %time model.fit(input_fn=input_fn, steps=2500)
          CPU times: user 11min 7s, sys: 1min 7s, total: 12min 15s
          Wall time: 4min 27s

Out[137]: DNNClassifier(params={'head':
           <tensorflow.contrib.learn.python.learn.estimators.head._MultiClassHead
           object at 0x116828cc0>, 'hidden_units': [500, 500, 500],
           'feature_columns': (_RealValuedColumn(column_name='lags', dimension=5,
           default_value=None, dtype=tf.float32, normalizer=None),), 'optimizer':
           None, 'activation_fn': <function relu at 0x1161441e0>, 'dropout':
           None, 'gradient_clip_norm': None, 'embedding_lr_multipliers': None,
           'input_layer_min_slice_size': None})

In [138]: data = test

In [139]: model.evaluate(input_fn=input_fn, steps=1)
Out[139]: {'loss': 0.82882184, 'accuracy': 0.48968107, 'global_step': 2500}

In [140]: pred = np.array(list(model.predict(input_fn=input_fn)))

In [141]: test['pos_dnn_tf'] = np.where(pred > 0, 1, -1)

In [142]: test['strat_dnn_tf'] = test['pos_dnn_tf'] * test['returns']

In [143]: test[['returns', 'strat_dnn_sk', 'strat_dnn_tf']].sum().apply(np.exp)
Out[143]: returns         0.878078
          strat_dnn_sk    1.242042
          strat_dnn_tf    1.063968
          dtype: float64

In [144]: test[['returns', 'strat_dnn_sk', 'strat_dnn_tf']].cumsum(
                     ).apply(np.exp).plot(figsize=(10, 6));
```

Figure 15-20. Performance of EUR/USD and DNN-based trading strategy (TensorFlow, randomized train-test split)

> **PERFORMANCE RESULTS**
>
> All performance results shown for the different algorithmic trading strategies from vectorized backtesting so far are illustrative only. Beyond the simplifying assumption of no transaction costs, the results depend on a number of other (mostly arbitrarily chosen) parameters. They also depend on the relative small end-of-day price data set used throughout for the EUR/USD exchange rate. The focus lies on illustrating the application of different approaches and ML algorithms to financial data, not on deriving a robust algorithmic trading strategy to be deployed in practice. The next chapter addresses some of these issues.

Conclusion

This chapter is about algorithmic trading strategies and judging their performance based on vectorized backtesting. It starts with a rather simple algorithmic trading strategy based on two simple moving averages, a type of strategy known and used in practice for decades. This strategy is used to illustrate vectorized backtesting, making heavy use of the vectorization capabilities of `NumPy` and `pandas` for data analysis.

Using OLS regression, the chapter also illustrates the random walk hypothesis

on the basis of a real financial time series. This is the benchmark against which any algorithmic trading strategy must prove its worth.

The core of the chapter is the application of machine learning algorithms, as introduced in "Machine Learning". A number of algorithms, the majority of which are of classification type, are used and applied based on mostly the same "rhythm." As features, lagged log returns data is used in a number of variants—although this is a restriction that for sure is not necessary. It is mostly done for convenience and simplicity. In addition, the analysis is based on a number of simplifying assumptions since the focus is mainly on the technical aspects of applying machine learning algorithms to financial time series data to predict the direction of financial market movements.

Further Resources

The papers referenced in this chapter are:

- Brock, William, Josef Lakonishok, and Blake LeBaron (1992). "Simple Technical Trading Rules and the Stochastic Properties of Stock Returns." *Journal of Finance*, Vol. 47, No. 5, pp. 1731–1764.
- Fama, Eugene (1965). "Random Walks in Stock Market Prices." Selected Papers, No. 16, Graduate School of Business, University of Chicago.
- Jensen, Michael (1978). "Some Anomalous Evidence Regarding Market Efficiency." *Journal of Financial Economics*, Vol. 6, No. 2/3, pp. 95–101.

Finance books covering topics relevant to this chapter include:

- Baxter, Martin, and Andrew Rennie (1996). *Financial Calculus*. Cambridge, England: Cambridge University Press.
- Chan, Ernest (2009). *Quantitative Trading*. Hoboken, NJ: John Wiley & Sons.
- Chan, Ernest (2013). *Algorithmic Trading*. Hoboken, NJ: John Wiley & Sons.

Chapter 16. Automated Trading

> *People worry that computers will get too smart and take over the world, but the real problem is that they're too stupid and they've already taken over the world.*
>
> —Pedro Domingos

"Now what?" one might think. A trading platform is available that allows one to retrieve historical data and streaming data, to place buy and sell orders, and to check the account status. A number of different methods have been introduced to derive algorithmic trading strategies by predicting the direction of market price movements. How can this all be put together to work in automated fashion? This question cannot be answered in any generality. However, this chapter addresses a number of topics that are important in this context. The chapter assumes that a single automated algorithmic trading strategy only shall be deployed. This simplifies, among others, aspects like capital and risk management.

The chapter covers the following topics:

"Capital Management"

: As this section demonstrates, depending on the strategy characteristics and the trading capital available, the Kelly criterion helps with sizing the trades.

"ML-Based Trading Strategy"

: To gain confidence in an algorithmic trading strategy, the strategy needs to be backtested thoroughly both with regard to performance and risk characteristics; the example strategy used is based on a classification algorithm from machine learning as introduced in Chapter 15.

"Online Algorithm"

: To deploy the algorithmic trading strategy for automated trading, it needs to be translated into an online algorithm that works with incoming streaming data in real time.

"Infrastructure and Deployment"

To run automated algorithmic trading strategies robustly and reliably, deployment in the cloud is the preferred option from an availability, performance, and security point of view.

"Logging and Monitoring"

To be able to analyze the history and certain events during the deployment of an automated trading strategy, logging plays an important role; monitoring via socket communication allows one to observe events (remotely) in real time.

Capital Management

A central question in algorithmic trading is how much capital to deploy to a given algorithmic trading strategy given the total available capital. The answer to this question depends on the main goal one is trying to achieve by algorithmic trading. Most individuals and financial institutions will agree that the *maximization of long-term wealth* is a good candidate objective. This is what Edward Thorpe had in mind when he derived the *Kelly criterion* for investing, as described in the paper by Rotando and Thorp (1992).

The Kelly Criterion in a Binomial Setting

The common way of introducing the theory of the Kelly criterion for investing is on the basis of a coin tossing game, or more generally a binomial setting (where only two outcomes are possible). This section follows that route. Assume a gambler is playing a coin tossing game against an infinitely rich bank or casino. Assume further that the probability for heads is some value p for which $\frac{1}{2} < p < 1$ holds. Probability for tails is defined by $q = 1 - p < \frac{1}{2}$. The gambler can place bets $b > 0$ of arbitrary size, whereby the gambler wins the same amount if right and loses it all if wrong. Given the assumptions about the probabilities, the gambler would of course want to bet on heads. Therefore, the expected value for this betting game B (i.e., the random variable representing this game) in a one-shot setting is:

$$\mathbf{E}[B] = p \cdot b - q \cdot b = (p - q) \cdot b > 0$$

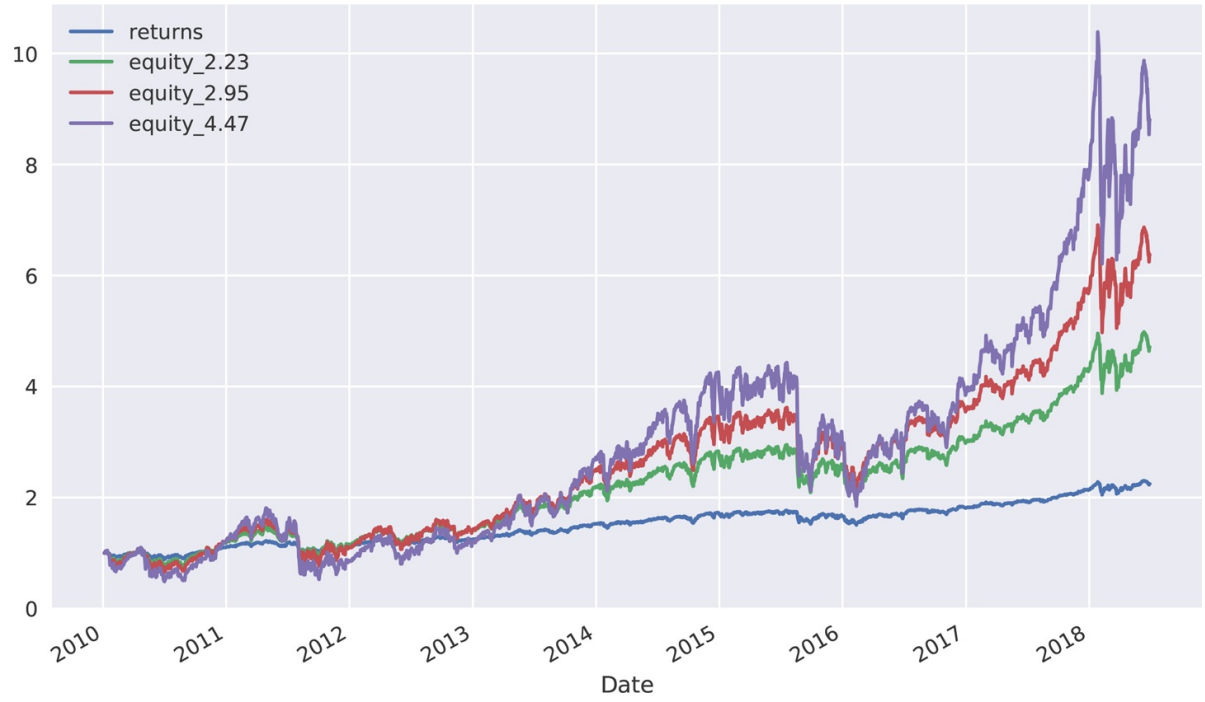

Figure 16-3. Cumulative performance of S&P 500 compared to equity position given different values of f

As Figure 16-3 illustrates, applying the optimal Kelly leverage leads to a rather erratic evolution of the equity position (high volatility) which is—given the leverage ratio of 4.47—intuitively plausible. One would expect the volatility of the equity position to increase with increasing leverage. Therefore, practitioners often reduce the leverage to, for example, "half Kelly"—i.e., in the current example to $\frac{1}{2} \cdot f^* \approx 2.23$. Therefore, Figure 16-3 also shows the evolution of the equity position of values lower than "full Kelly." The risk indeed reduces with lower values of *f*.

ML-Based Trading Strategy

Chapter 14 introduces the FXCM trading platform, its REST API, and the Python wrapper package `fxcmpy`. This section combines an ML-based approach for predicting the direction of market price movements with historical data from the FXCM REST API to backtest an algorithmic trading strategy for the `EUR/USD` currency pair. It uses vectorized backtesting, taking into account this time the bid-ask spread as proportional transaction costs. It also adds, compared to the plain vectorized backtesting approach as introduced in Chapter 15, a more in-depth analysis of the risk characteristics of the trading strategy tested.

Vectorized Backtesting

The backtest is based on intraday data, more specifically on bars of length five minutes. The following code connects to the FXCM REST API and retrieves five-minute bar data for a whole month. Figure 16-4 visualizes the mid close prices over the period for which data is retrieved:

```
In [36]: import fxcmpy

In [37]: fxcmpy.__version__
Out[37]: '1.1.33'

In [38]: api = fxcmpy.fxcmpy(config_file='../fxcm.cfg')  ❶

In [39]: data = api.get_candles('EUR/USD', period='m5',
                                start='2018-06-01 00:00:00',
                                stop='2018-06-30 00:00:00')  ❶

In [40]: data.iloc[-5:, 4:]
Out[40]:                      askopen  askclose  askhigh   asklow  tickqty
         date
         2018-06-29 20:35:00  1.16862   1.16882  1.16896  1.16839      601
         2018-06-29 20:40:00  1.16882   1.16853  1.16898  1.16852      387
         2018-06-29 20:45:00  1.16853   1.16826  1.16862  1.16822      592
         2018-06-29 20:50:00  1.16826   1.16836  1.16846  1.16819      842
         2018-06-29 20:55:00  1.16836   1.16861  1.16876  1.16834      540

In [41]: data.info()
         <class 'pandas.core.frame.DataFrame'>
         DatetimeIndex: 6083 entries, 2018-06-01 00:00:00 to 2018-06-29 20:55:00
         Data columns (total 9 columns):
         bidopen     6083 non-null float64
         bidclose    6083 non-null float64
         bidhigh     6083 non-null float64
         bidlow      6083 non-null float64
         askopen     6083 non-null float64
         askclose    6083 non-null float64
         askhigh     6083 non-null float64
         asklow      6083 non-null float64
         tickqty     6083 non-null int64
         dtypes: float64(8), int64(1)
         memory usage: 475.2 KB

In [42]: spread = (data['askclose'] - data['bidclose']).mean()  ❷
         spread  ❷
Out[42]: 2.6338977478217845e-05

In [43]: data['midclose'] = (data['askclose'] + data['bidclose']) / 2  ❸
```

```
In [44]: ptc = spread / data['midclose'].mean()  ❹
         ptc  ❹
Out[44]: 2.255685318140426e-05

In [45]: data['midclose'].plot(figsize=(10, 6), legend=True);
```

❶ Connects to the API and retrieves the data.
❷ Calculates the average bid-ask spread.
❸ Calculates the mid close prices from the ask and bid close prices.
❹ Calculates the average proportional transaction costs given the average spread and the average mid close price.

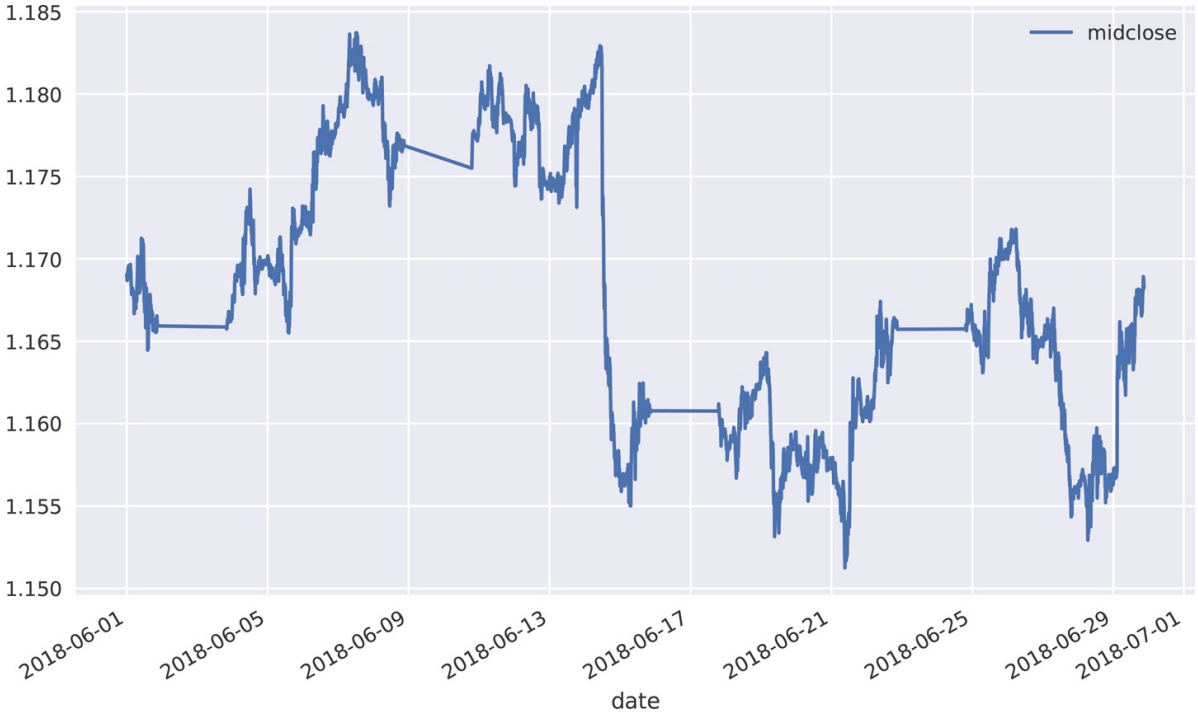

Figure 16-4. EUR/USD exchange rate (five-minute bars)

The ML-based strategy is based on lagged return data that is binarized. In other words, the ML algorithm learns from historical patterns of upward and downward movements whether another upward or downward movement is more likely. Accordingly, the following code creates features data with values of `0` and `1` as well as labels data with values of `+1` and `-1` indicating the observed market direction in all cases:

```
In [46]: data['returns'] = np.log(data['midclose'] / data['midclose'].shift(1))
```

```
In [47]: data.dropna(inplace=True)

In [48]: lags = 5

In [49]: cols = []
         for lag in range(1, lags + 1):
             col = 'lag_{}'.format(lag)
             data[col] = data['returns'].shift(lag)  ❶
             cols.append(col)

In [50]: data.dropna(inplace=True)

In [51]: data[cols] = np.where(data[cols] > 0, 1, 0)  ❷

In [52]: data['direction'] = np.where(data['returns'] > 0, 1, -1)  ❸

In [53]: data[cols + ['direction']].head()
Out[53]:                      lag_1  lag_2  lag_3  lag_4  lag_5  direction
         date
         2018-06-01 00:30:00      1      0      1      0      1          1
         2018-06-01 00:35:00      1      1      0      1      0          1
         2018-06-01 00:40:00      1      1      1      0      1          1
         2018-06-01 00:45:00      1      1      1      1      0          1
         2018-06-01 00:50:00      1      1      1      1      1         -1
```

❶ Creates the lagged return data given the number of lags.

❷ Transforms the feature values to binary data.

❸ Transforms the returns data to directional label data.

Given the features and label data, different supervised learning algorithms can now be applied. In what follows, a support vector machine algorithm for classification is used from the `scikit-learn` ML package. The code trains and tests the algorithmic trading strategy based on a sequential train-test split. The accuracy scores of the model for the training and test data are slightly above 50%, while the score is even a bit higher on the test data. Instead of accuracy scores, one would also speak in a financial trading context of the *hit ratio* of the trading strategy; i.e., the number of winning trades compared to all trades. Since the hit ratio is greater than 50%, this might indicate—in the context of the Kelly criterion—a slight edge compared to a random walk setting:

```
In [54]: from sklearn.svm import SVC
         from sklearn.metrics import accuracy_score
```

```
In [55]: model = SVC(C=1, kernel='linear', gamma='auto')

In [56]: split = int(len(data) * 0.80)

In [57]: train = data.iloc[:split].copy()

In [58]: model.fit(train[cols], train['direction'])
Out[58]: SVC(C=1, cache_size=200, class_weight=None, coef0=0.0,
           decision_function_shape='ovr', degree=3, gamma='auto', kernel='linear',
             max_iter=-1, probability=False, random_state=None, shrinking=True,
             tol=0.001, verbose=False)

In [59]: accuracy_score(train['direction'], model.predict(train[cols]))  ❶
Out[59]: 0.5198518823287389

In [60]: test = data.iloc[split:].copy()

In [61]: test['position'] = model.predict(test[cols])

In [62]: accuracy_score(test['direction'], test['position'])  ❷
Out[62]: 0.5419407894736842
```

❶ The accuracy of the predictions from the trained model *in-sample* (training data).

❷ The accuracy of the predictions from the trained model *out-of-sample* (test data).

It is well known that the hit ratio is only one aspect of success in financial trading. Also crucial are, among other things, the transaction costs implied by the trading strategy and getting the important trades right.[2] To this end, only a formal vectorized backtesting approach allows judgment of the quality of the trading strategy. The following code takes into account the proportional transaction costs based on the average bid-ask spread. Figure 16-5 compares the performance of the algorithmic trading strategy (without and with proportional transaction costs) to the performance of the passive benchmark investment:

```
In [63]: test['strategy'] = test['position'] * test['returns']  ❶

In [64]: sum(test['position'].diff() != 0)  ❷
Out[64]: 660

In [65]: test['strategy_tc'] = np.where(test['position'].diff() != 0,
                                        test['strategy'] - ptc,  ❸
                                        test['strategy'])
```

```
In [66]: test[['returns', 'strategy', 'strategy_tc']].sum(
                 ).apply(np.exp)
Out[66]: returns        0.999324
         strategy       1.026141
         strategy_tc    1.010977
         dtype: float64

In [67]: test[['returns', 'strategy', 'strategy_tc']].cumsum(
                 ).apply(np.exp).plot(figsize=(10, 6));
```

❶ Derives the log returns for the ML-based algorithmic trading strategy.

❷ Calculates the number of trades implied by the trading strategy based on changes in the position.

❸ Whenever a trade takes place, the proportional transaction costs are subtracted from the strategy's log return on that day.

Figure 16-5. Performance of EUR/USD exchange rate and algorithmic trading strategy

LIMITATIONS OF VECTORIZED BACKTESTING

Vectorized backtesting has its limits with regard to how closely to market realities strategies can be tested. For example, it does not allow direct inclusion of fixed transaction costs per trade. One could, as an approximation, take a multiple of the average proportional transaction

costs (based on average position sizes) to account indirectly for fixed transactions costs. However, this would not be precise in general. If a higher degree of precision is required other approaches, such as *event-based backtesting* with explicit loops over every bar of the price data, need to be applied.

Optimal Leverage

Equipped with the trading strategy's log returns data, the mean and variance values can be calculated in order to derive the optimal leverage according to the Kelly criterion. The code that follows scales the numbers to annualized values, although this does not change the optimal leverage values according to the Kelly criterion since the mean return and the variance scale with the same factor:

```
In [68]: mean = test[['returns', 'strategy_tc']].mean() * len(data) * 12  ❶
         mean
Out[68]: returns       -0.040535
         strategy_tc    0.654711
         dtype: float64

In [69]: var = test[['returns', 'strategy_tc']].var() * len(data) * 12  ❷
         var
Out[69]: returns        0.007861
         strategy_tc    0.007837
         dtype: float64

In [70]: vol = var ** 0.5  ❸
         vol
Out[70]: returns        0.088663
         strategy_tc    0.088524
         dtype: float64

In [71]: mean / var  ❹
Out[71]: returns        -5.156448
         strategy_tc    83.545792
         dtype: float64

In [72]: mean / var * 0.5  ❺
Out[72]: returns        -2.578224
         strategy_tc    41.772896
         dtype: float64
```

❶ Annualized mean returns.

❷ Annualized variances.

❸ Annualized volatilities.

Conclusion

This chapter is about the deployment of an algorithmic trading strategy—based on a classification algorithm from machine learning to predict the direction of market movements—in automated fashion. It addresses such important topics as capital management (based on the Kelly criterion), vectorized backtesting for performance and risk, the transformation of offline to online trading algorithms, an appropriate infrastructure for deployment, as well as logging and monitoring during deployment.

The topic of this chapter is complex and requires a broad skill set from the algorithmic trading practitioner. On the other hand, having a REST API for algorithmic trading available, such as the one from FXCM, simplifies the automation task considerably since the core part boils down mainly to making use of the capabilities of the Python wrapper package `fxcmpy` for tick data retrieval and order placement. Around this core, elements to mitigate operational and technical risks as far as possible have to be added.

Python Scripts

Automated Trading Strategy

The following is the Python script to implement the algorithmic trading strategy in automated fashion, including logging and monitoring.

```
#
# Automated ML-Based Trading Strategy for FXCM
# Online Algorithm, Logging, Monitoring
#
# Python for Finance, 2nd ed.
# (c) Dr. Yves J. Hilpisch
#
import zmq
import time
import pickle
import fxcmpy
import numpy as np
import pandas as pd
import datetime as dt
```

```
sel = ['tradeId', 'amountK', 'currency',
       'grossPL', 'isBuy']

log_file = 'automated_strategy.log'

# loads the persisted algorithm object
algorithm = pickle.load(open('algorithm.pkl', 'rb'))

# sets up the socket communication via ZeroMQ (here: "publisher")
context = zmq.Context()
socket = context.socket(zmq.PUB)

# this binds the socket communication to all IP addresses of the machine
socket.bind('tcp://0.0.0.0:5555')


def logger_monitor(message, time=True, sep=True):
    ''' Custom logger and monitor function.
    '''
    with open(log_file, 'a') as f:
        t = str(dt.datetime.now())
        msg = ''
        if time:
            msg += '\n' + t + '\n'
        if sep:
            msg += 66 * '=' + '\n'
        msg += message + '\n\n'
        # sends the message via the socket
        socket.send_string(msg)
        # writes the message to the log file
        f.write(msg)


def report_positions(pos):
    ''' Prints, logs and sends position data.
    '''
    out = '\n\n' + 50 * '=' + '\n'
    out += 'Going {}.\n'.format(pos) + '\n'
    time.sleep(2)  # waits for the order to be executed
    out += str(api.get_open_positions()[sel]) + '\n'
    out += 50 * '=' + '\n'
    logger_monitor(out)
    print(out)


def automated_strategy(data, dataframe):
    ''' Callback function embodying the trading logic.
    '''
    global min_bars, position, df
```

```
# resampling of the tick data
df = dataframe.resample(bar, label='right').last().ffill()

if len(df) > min_bars:
    min_bars = len(df)
    logger_monitor('NUMBER OF TICKS: {} | '.format(len(dataframe)) +
                   'NUMBER OF BARS: {}'.format(min_bars))
    # data processing and feature preparation
    df['Mid'] = df[['Bid', 'Ask']].mean(axis=1)
    df['Returns'] = np.log(df['Mid'] / df['Mid'].shift(1))
    df['Direction'] = np.where(df['Returns'] > 0, 1, -1)
    # picks relevant points
    features = df['Direction'].iloc[-(lags + 1):-1]
    # necessary reshaping
    features = features.values.reshape(1, -1)
    # generates the signal (+1 or -1)
    signal = algorithm.predict(features)[0]

    # logs and sends major financial information
    logger_monitor('MOST RECENT DATA\n' +
                   str(df[['Mid', 'Returns', 'Direction']].tail()),
                   False)
    logger_monitor('features: ' + str(features) + '\n' +
                   'position: ' + str(position) + '\n' +
                   'signal:   ' + str(signal), False)

    # trading logic
    if position in [0, -1] and signal == 1:  # going long?
        api.create_market_buy_order(
            symbol, size - position * size)  # places a buy order
        position = 1  # changes position to long
        report_positions('LONG')

    elif position in [0, 1] and signal == -1:  # going short?
        api.create_market_sell_order(
            symbol, size + position * size)  # places a sell order
        position = -1  # changes position to short
        report_positions('SHORT')
    else:  # no trade
        logger_monitor('no trade placed')

    logger_monitor('****END OF CYCLE***\n\n', False, False)

if len(dataframe) > 350:  # stopping condition
    api.unsubscribe_market_data('EUR/USD')  # unsubscribes from data stream
    report_positions('CLOSE OUT')
    api.close_all()  # closes all open positions
    logger_monitor('***CLOSING OUT ALL POSITIONS***')
```

```
if __name__ == '__main__':
    symbol = 'EUR/USD'  # symbol to be traded
    bar = '15s'  # bar length; adjust for testing and deployment
    size = 100  # position size in thousand currency units
    position = 0  # initial position
    lags = 5  # number of lags for features data
    min_bars = lags + 1  # minimum length for resampled DataFrame
    df = pd.DataFrame()
    # adjust configuration file location
    api = fxcmpy.fxcmpy(config_file='../fxcm.cfg')
    # the main asynchronous loop using the callback function
    api.subscribe_market_data(symbol, (automated_strategy,))
```

Strategy Monitoring

The following is the Python script to implement a local or remote monitoring of the automated algorithmic trading strategy via socket communication.

```
#
# Automated ML-Based Trading Strategy for FXCM
# Strategy Monitoring via Socket Communication
#
# Python for Finance, 2nd ed.
# (c) Dr. Yves J. Hilpisch
#
import zmq

# sets up the socket communication via ZeroMQ (here: "subscriber")
context = zmq.Context()
socket = context.socket(zmq.SUB)

# adjust the IP address to reflect the remote location
socket.connect('tcp://REMOTE_IP_ADDRESS:5555')

# configures the socket to retrieve every message
socket.setsockopt_string(zmq.SUBSCRIBE, '')

while True:
    msg = socket.recv_string()
    print(msg)
```

Further Resources

The papers cited in this chapter are:

- Rotando, Louis, and Edward Thorp (1992). “The Kelly Criterion and the Stock Market.” *The American Mathematical Monthly*, Vol. 99, No. 10, pp. 922–931.
- Hung, Jane (2010): “Betting with the Kelly Criterion.” *http://bit.ly/betting_with_kelly*.

For a comprehensive online training program covering Python for algorithmic trading see *http://certificate.tpq.io*.

---

1 The exposition follows Hung (2010).

2 It is a stylized empirical fact that it is of paramount importance for investment and trading performance to get the largest market movements right—i.e., the biggest upward *and* downward movements. This aspect is neatly illustrated in Figures 16-5 and 16-7, which show that the trading strategy gets a large upward movement in the underlying instrument wrong, leading to a large dip for the trading strategy.

3 Leverage increases risks associated with trading strategies significantly. Traders should read the risk disclaimers and regulations carefully. A positive backtesting performance is also no guarantee whatsoever of future performance. All results shown are illustrative only and are meant to demonstrate the application of programming and analytics approaches. In some jurisdictions, such as in Germany, leverage ratios are capped for retail traders based on different groups of financial instruments.

4 Use the link *http://bit.ly/do_sign_up* to get a 10 USD bonus on DigitalOcean when signing up for a new account.

5 Note that the socket communication as implemented in the two scripts is not encrypted and is sending plain text over the web, which might represent a security risk in production.