Extracted from:

Python Companion to Data Science

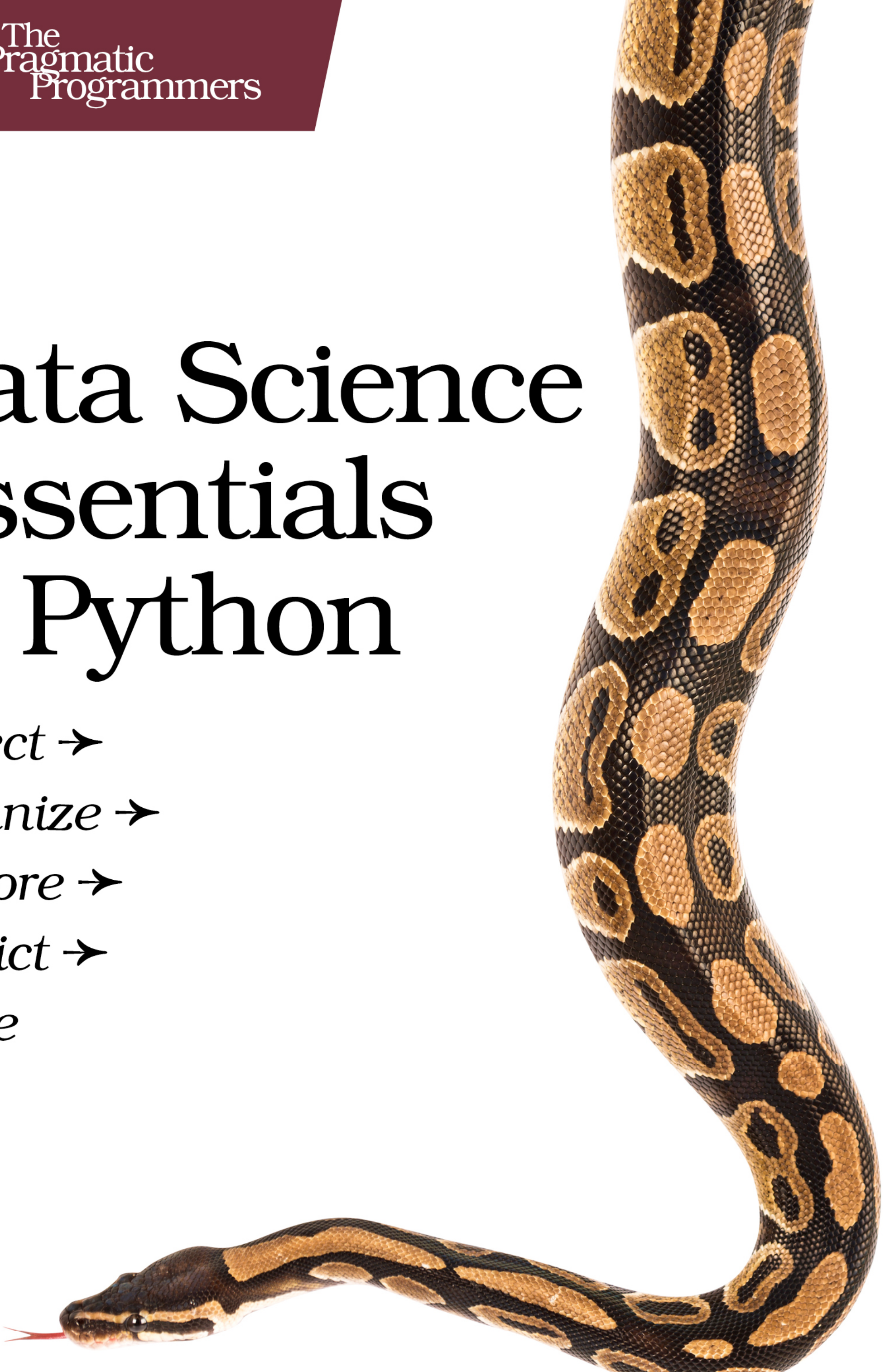
Collect → Organize → Explore → Predict → Value

The Pragmatic Bookshelf
Raleigh, North Carolina

The Pragmatic Programmers

Data Science Essentials in Python

Collect →
Organize →
Explore →
Predict →
Value



Dmitry Zinoviev
edited by Katharine Dvorak

Python Companion to Data Science

Collect → Organize → Explore → Predict → Value

Dmitry Zinoviev

The Pragmatic Bookshelf
Raleigh, North Carolina

The team that produced this book includes:

Katharine Dvorak (editor)
Potomac Indexing, LLC (index)
Nicole Abramowitz (copyedit)
Gilson Graphics (layout)
Janet Furlow (producer)

Printed in the United States of America.
ISBN-13: 978-1-68050-184-1
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—August 2016

To my beautiful and most intelligent wife Anna; to our children: graceful ballerina Eugenia and romantic gamer Roman; and to my first data science class of summer 2015.

It is a capital mistake to theorize before one has data.

➤ *Sir Arthur Conan Doyle, British writer*

CHAPTER 5

Working with Tabular Numeric Data

Often raw data comes from all kinds of text documents. Quite often the text actually represents numbers. Excel and CSV spreadsheets and especially database tables may contain millions or billions of numerical records. Core Python is an excellent text-processing tool, but it sometimes fails to deliver adequate numeric performance. That's where numpy comes to the rescue.

NumPy—Numeric Python (imported as numpy)—is an interface to a family of efficient and parallelizable functions that implement high-performance numerical operations. The module numpy provides a new Python data structure —array—and a toolbox of array-specific functions, as well as support for random numbers, data aggregation, linear algebra, Fourier transform, and other goodies.

Bridge to Terabytia

If your program needs access to huge amounts of numerical data —terabytes and more—you can't avoid using the module h5py.[1] The module is a portal to the HDF5 binary data format that works with a lot of third-party software, such as IDL and MATLAB. h5py imitates familiar numpy and Python mechanisms, such as arrays and dictionaries. Once you know how to use numpy, you can go straight to h5py—but not in this book.

In this chapter, you'll learn how to create numpy arrays of different shapes and from different sources, reshape and slice arrays, add array indexes, and apply arithmetic, logic, and aggregation functions to some or all array elements.

1. www.h5py.org

Unit 21

Creating Arrays

numpy arrays are more compact and faster than native Python lists, especially in multidimensional cases. However, unlike lists, arrays are homogeneous: you cannot mix and match array items that belong to different data types.

There are several ways to create a numpy array. The function array() creates an array from array-like data. The data can be a list, a tuple, or another array. numpy infers the type of the array elements from the data, unless you explicitly pass the dtype parameter. numpy supports close to twenty data types, such as bool_, int64, uint64, float64, and <U32 (for Unicode strings).

When numpy creates an array, it doesn't copy the data from the source to the new array, but it links to it for efficiency reasons. This means that, by default, a numpy array is a view of its underlying data, not a copy of it. If the underlying data object changes, the array data changes, too. If this behavior is undesirable (which it always is, unless the amount of data to copy is prohibitively large), pass copy=True to the constructor.

Lists Are Arrays, and Arrays Are Arrays, Too

Contrary to their name, Python "lists" are actually implemented as arrays, not as lists. No storage is reserved for forward pointers, because no forward pointers are used. Large Python lists require only about 13% more storage than "real" numpy arrays. However, Python executes some simple built-in operations, such as sum(), five to ten times faster on lists than on arrays. Ask yourself if you really need any numpy-specific features before you start a numpy project!

Let's create our first array—a silly array of the first ten positive integer numbers:

```
import numpy as np
numbers = np.array(range(1, 11), copy=True)
```

```
⇒ array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
```

The functions ones(), zeros(), and empty() construct arrays of all ones, all zeros, and all uninitialized entries, respectively. They then take a required shape parameter—a list or tuple of array dimensions.

```
ones = np.ones([2, 4], dtype=np.float64)
```

```
⇛ array([[ 1.,  1.,  1.,  1.],
⇛        [ 1.,  1.,  1.,  1.]])
```

```
zeros = np.zeros([2, 4], dtype=np.float64)
```

```
⇛ array([[ 0.,  0.,  0.,  0.],
⇛        [ 0.,  0.,  0.,  0.]])
```

```
empty = np.empty([2, 4], dtype=np.float64)
# The array content is not always zeros!
```

```
⇛ array([[ 0.,  0.,  0.,  0.],
⇛        [ 0.,  0.,  0.,  0.]])
```

numpy stores the number of dimensions, the shape, and the data type of an array in the attributes ndim, shape, and dtype.

```
ones.shape # The original shape, unless reshaped
```

```
⇛ (2, 4)
```

```
numbers.ndim # Same as len(numbers.shape)
```

```
⇛ 1
```

```
zeros.dtype
```

```
⇛ dtype('float64')
```

The function eye(N, M=None, k=0, dtype=np.float) constructs an N×M *eye*-dentity matrix with ones on the kth diagonal and zeros elsewhere. Positive k denotes the diagonals above the main diagonal. When M is None (default), M=N.

```
eye = np.eye(3, k=1)
```

```
⇛ array([[ 0.,  1.,  0.],
⇛        [ 0.,  0.,  1.],
⇛        [ 0.,  0.,  0.]])
```

When you need to multiply several matrices, use an identity matrix as the initial value of the accumulator in the multiplication chain.

In addition to the good old built-in range(), numpy has its own, more efficient way of generating arrays of regularly spaced numbers: the function arange([start,] stop[, step,], dtype=None).

```
np_numbers = np.arange(2, 5, 0.25)
```

```
⇒ array([ 2.  ,  2.25,  2.5 ,  2.75,  3.  ,  3.25,  3.5 ,  3.75,  4.  ,
⇒         4.25,  4.5 ,  4.75])
```

Just like with range(), the value of stop can be smaller than start—but then step must be negative, and the order of numbers in the array is decreasing.

numpy records the type of items at the time of array creation, but the type is not immutable: you can change it later by calling the astype(dtype, casting="unsafe", copy=True) function. In the case of type narrowing (converting to a more specific data type), some information may be lost. This is true about any narrowing, not just in numpy.

```
np_inumbers = np_numbers.astype(np.int)
```

```
⇒ array([2, 2, 2, 2, 3, 3, 3, 3, 4, 4, 4, 4])
```

Most numpy operations (such as transposing, which is discussed in the next unit on page 11) return a view, not a copy of the original array. To preserve original data, the function copy() creates a copy of an existing array. Any changes to the original array don't affect the copy, but if your array has one billion items, think twice before copying it.

```
np_inumbers_copy = np_inumbers.copy()
```

Let's now proceed to more advanced operations.

Unit 22

Transposing and Reshaping

Unlike the great monuments of the past, numpy arrays are not carved in stone. They can easily change their shape and orientation without being called opportunists. Let’s build a one-dimensional array of some S&P stock symbols and twist it in every possible way:

```
# Some S&P stock symbols
sap = np.array(["MMM", "ABT", "ABBV", "ACN", "ACE", "ATVI", "ADBE", "ADT"])
```

```
⇒ array(['MMM', 'ABT', 'ABBV', 'ACN', 'ACE', 'ATVI', 'ADBE', 'ADT'],
⇒       dtype='<U4')
```

The function reshape(d0, d1, ...) changes the shape of an existing array. The arguments define the new dimensions. The total number of items in the old and new shapes must be equal: the conservation law still holds in numpyland!

```
sap2d = sap.reshape(2, 4)
```

```
⇒ array([['MMM', 'ABT', 'ABBV', 'ACN'],
⇒        ['ACE', 'ATVI', 'ADBE', 'ADT']],
⇒       dtype='<U4')
```

```
sap3d = sap.reshape(2, 2, 2)
```

```
⇒ array([[['MMM', 'ABT'],
⇒         ['ABBV', 'ACN']],
⇒
⇒        [['ACE', 'ATVI'],
⇒         ['ADBE', 'ADT']]],
⇒       dtype='<U4')
```

To transpose an array, you don’t even need to call a function: the value of the attribute T is the transposed view of the array (for a one-dimensional array, data.T==data; for a two-dimensional array, the rows and the columns are swapped).

```
sap2d.T
```

```
⇒ array([['MMM', 'ACE'],
⇒        ['ABT', 'ATVI'],
⇒        ['ABBV', 'ADBE'],
⇒        ['ACN', 'ADT']],
⇒       dtype='<U4')
```

Essentially, the attribute T shows us the matrix through a relabeling cross-hair: axis number 0 ("vertical") becomes axis number 1 ("horizontal"), and the other way around. The function swapaxes() is a more general version of the T. It transposes a multidimensional array by swapping any two axes that you pass as the parameters. Naturally, passing the axes 0 and 1 for a two-dimensional array simply transposes the array, just as before.

```
sap3d.swapaxes(1, 2)
```

```
⇒ array([[['MMM', 'ABBV'],
⇒         ['ABT', 'ACN']],
⇒
⇒        [['ACE', 'ADBE'],
⇒         ['ATVI', 'ADT']]],
⇒       dtype='<U4')
```

The function transpose() is even more general than swapaxes() (despite its name implying similarity to the T attribute). transpose() permutes some or all axes of a multidimensional array according to its parameter, which must be a tuple. In the following example, the first axis remains "vertical," but the other two axes are swapped.

```
sap3d.transpose((0, 2, 1))
```

```
⇒ array([[['MMM', 'ABBV'],
⇒         ['ABT', 'ACN']],
⇒
⇒        [['ACE', 'ADBE'],
⇒         ['ATVI', 'ADT']]],
⇒       dtype='<U4')
```

Incidentally, the result is the same as for swapaxes(1, 2)!

Unit 23

Indexing and Slicing

numpy arrays support the same indexing [i] and slicing [i:j] operations as Python lists. In addition, they implement Boolean indexing: you can use an array of Boolean values as an index, and the result of the selection is the array of items of the original array for which the Boolean index is True. Often the Boolean array is in turn a result of broadcasting. You can use Boolean indexing on the right-hand side (for selection) and on the left-hand side (for assignment).

Suppose your data sponsor told you that all data in the data set dirty is strictly non-negative. This means that any negative value is not a true value but an error, and you must replace it with something that makes more sense—say, with a zero. This operation is called *data cleaning*. To clean the dirty data, locate the offending values and substitute them with reasonable alternatives.

```
dirty = np.array([9, 4, 1, -0.01, -0.02, -0.001])
whos_dirty = dirty < 0 # Boolean array, to be used as Boolean index
```

```
⇛ array([False, False, False,  True,  True,  True], dtype=bool)
```

```
dirty[whos_dirty] = 0 # Change all negative values to 0
```

```
⇛ array([9, 4, 1, 0, 0, 0])
```

You can combine several Boolean expressions with the operators | (logical or), & (logical and), and - (logical not). Which of the items in the following list are between -0.5 and 0.5? Ask numpy!

```
linear = np.arange(-1, 1.1, 0.2)
(linear <= 0.5) & (linear >= -0.5)
```

```
⇛ array([False, False, False,  True,  True,  True,  True,  True, False,
⇛        False, False], dtype=bool)
```

Boolean and "Boolean"

Relational operators (such as < and ==) have lower precedence than the bit-wise operators &, |, and ! that represent "Boolean" operators on numpy arrays. This is very confusing, because "normal" Python Boolean operators—or, and, and not—have lower precedence than the relational operators. You must enclose array comparisons in parentheses to ensure that numpy evaluates them first.

Another cool feature of numpy arrays is "smart" indexing and "smart" slicing, whereby an index is not a scalar but an array or list of indexes. The result of the selection is the array of items that are referenced in the index. Let's select the second, the third, and the last stock symbols from our S&P list. (That's "smart" indexing.)

```
sap[[1, 2, -1]]
```

```
⇒ array(['ABT', 'BBV', 'ADT'],
⇒       dtype='<U4')
```

Why not extract all rows in the middle column from the reshaped array? (That's "smart" slicing.) In fact, you can do it two ways:

```
sap2d[:, [1]]
```

```
⇒ array([['ABT'],
⇒         ['ATVI']],
⇒       dtype='<U4')
```

```
sap2d[:, 1]
```

```
⇒ array(['ABT', 'ATVI'],
⇒       dtype='<U4')
```

Python is famous for giving us many similar problem-solving tools and not forgiving us for choosing a wrong one. Compare the two selections shown earlier. The first selection is a two-dimensional matrix; the second one is a one-dimensional array. Depending on what you wanted to extract, one of them is wrong. Oops. Make sure you check that you've gotten what you wanted.