Extracted from:

Python Companion to Data Science

Collect → Organize → Explore → Predict → Value

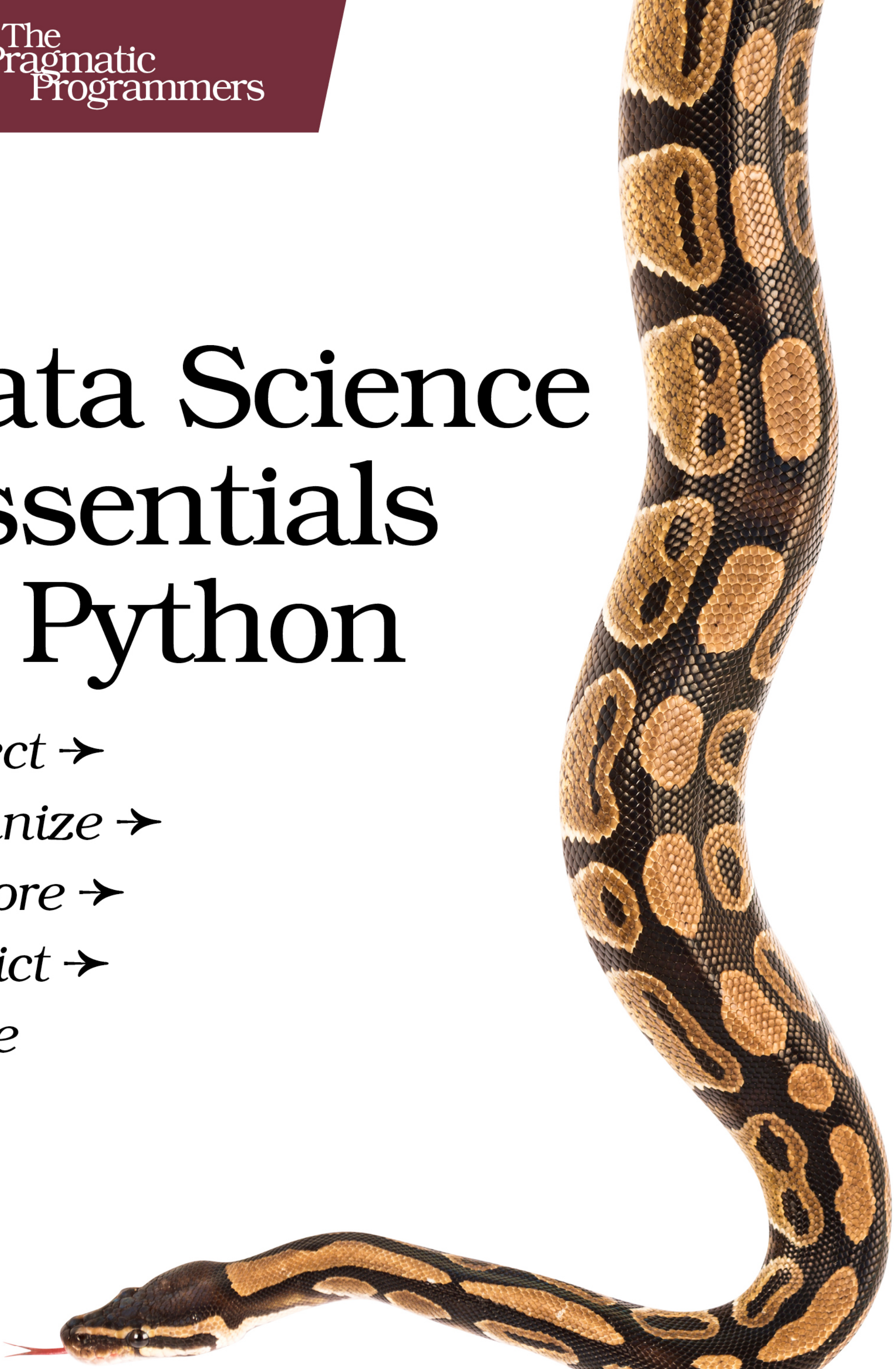
The Pragmatic Bookshelf

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.

And I spoke to them in as many languages as I had the least smattering of, which were High and Low Dutch, Latin, French, Spanish, Italian, and Lingua Franca, but all to no purpose.

➤ *Jonathan Swift, Anglo-Irish satirist*

CHAPTER 2

Core Python for Data Science

Some features of the core Python language are more important for data analysis than others. In this chapter, you'll look at the most essential of them: string functions, data structures, list comprehension, counters, file and web functions, regular expressions, globbing, and data pickling. You'll learn how to use Python to extract data from local disk files and the Internet, store them into appropriate data structures, locate bits and pieces matching certain patterns, and serialize and de-serialize Python objects for future processing. However, these functions are by no means specific to data science or data analysis tasks and are found in many other applications.

It's a common misunderstanding that the presence of high-level programming tools makes low-level programming obsolete. With the Anaconda distribution of Python alone providing more than 350 Python packages, who needs to split strings and open files? The truth is, there are at least as many non-standard data sources in the world as those that follow the rules.

All standard data frames, series, CSV readers, and word tokenizers follow the rules set up by their creators. They fail miserably when they come across anything out of compliance with the rules. That's when you blow the dust off this book and demote yourself from glorified data scientist to humble but very useful computer programmer.

You may need to go as far "down" as to the string functions—in fact, they are just around the corner on page 8.

Unit 4

Understanding Basic String Functions

A string is a basic unit of interaction between the world of computers and the world of humans. Initially, almost all raw data is stored as strings. In this unit, you'll learn how to assess and manipulate text strings.

All functions described in this unit are members of the str built-in class.

The *case conversion* functions return a copy of the original string s: lower() converts all characters to lowercase; upper() converts all characters to uppercase; and capitalize() converts the first character to uppercase and all other characters to lowercase. These functions don't affect non-alphabetic characters. Case conversion functions are an important element of normalization, which you'll look at on page ?.

The *predicate* functions return True or False, depending on whether the string s belongs to the appropriate class: islower() checks if all alphabetic characters are in lowercase; isupper() checks if all alphabetic characters are in uppercase; isspace() checks if all characters are spaces; isdigit() checks if all characters are decimal digits in the range 0–9; and isalpha() checks if all characters are alphabetic characters in the ranges a–z or A–Z. You will use these functions to recognize valid words, nonnegative integer numbers, punctuation, and the like.

Sometimes Python represents string data as raw binary arrays, not as character strings, especially when the data came from an external source: an external file, a database, or the web. Python uses the b notation for binary arrays. For example, bin = b"Hello" is a binary array; s = "Hello" is a string. Respectively, s[0] is 'H' and bin[0] is 72, where 72 is the ASCII charcode for the character 'H'. The *decoding* functions convert a binary array to a character string and back: bin.decode() converts a binary array to a string, and s.encode() converts a string to a binary array. Many Python functions expect that binary data is converted to strings until it is further processed.

The first step toward string processing is getting rid of unwanted whitespaces (including new lines and tabs). The functions lstrip() (left strip), rstrip() (right strip), and strip() remove all whitespaces at the beginning, at the end, or all around the string. (They don't remove the inner spaces.) With all these removals, you should be prepared to end up with an empty string!

```
"    Hello,  world!  \t\t\n".strip()
```

```
⇒ 'Hello,  world!'
```

Often a string consists of several tokens, separated by delimiters such as spaces, colons, and commas. The function split(delim='') splits the string s into a list of substrings, using delim as the delimiter. If the delimiter isn't specified, Python splits the string by all whitespaces and lumps all contiguous whitespaces together:

```
"Hello,  world!".split() # Two spaces!
```

```
⇒ ['Hello,', 'world!']
```

```
"Hello,  world!".split(" ") # Two spaces!
```

```
⇒ ['Hello,', '', 'world!']
```

```
"www.networksciencelab.com".split(".")
```

```
⇒ ['www', 'networksciencelab', 'com']
```

The sister function join(ls) joins a list of strings ls into one string, using the object string as the glue. You can recombine fragments with join():

```
", ".join(["alpha", "bravo", "charlie", "delta"])
```

```
⇒ 'alpha, bravo, charlie, delta'
```

In the previous example, join() inserts the glue only between the strings and not in front of the first string or after the last string. The result of splitting a string and joining the fragments again is often indistinguishable from replacing the split delimiter with the glue:

```
"-".join("1.617.305.1985".split("."))
```

```
⇒ '1-617-305-1985'
```

Sometimes you may want to use the two functions together to remove unwanted whitespaces from a string. You can accomplish the same effect by regular expression-based substitution (which you'll look at later on page ?).

```
" ".join("This string\n\r  has    many\t\tspaces".split())
```

```
⇒ 'This string has many spaces'
```

The function find(needle) returns the index of the first occurrence of the substring needle in the object string or -1 if the substring is not present. This function is case-sensitive. It is used to find a fragment of interest in a string —if it exists.

```
"www.networksciencelab.com".find(".com")
```

```
⇒ 21
```

The function count(needle) returns the number of non-overlapping occurrences of the substring needle in the object string. This function is also case-sensitive.

```
"www.networksciencelab.com".count(".")
```

```
⇒ 2
```

Strings are an important building block of any data-processing program, but not the only building block—and not the most efficient building block, either. You will also use lists, tuples, sets, and dictionaries to bundle string and numeric data and enable efficient searching and sorting.

Unit 5

Choosing the Right Data Structure

The most commonly used compound data structures in Python are lists, tuples, sets, and dictionaries. All four of them are collections.

Python implements *lists* as arrays. They have linear search time, which makes them impractical for storing large amounts of searchable data.

Tuples are immutable lists. Once created, they cannot be changed. They still have linear search time.

Unlike lists and tuples, *sets* are not sequences: set items don't have indexes. Sets can store at most one copy of an item and have sublinear O(log(N)) search time. They are excellent for membership look-ups and eliminating duplicates (if you convert a list with duplicates to a set, the duplicates are gone):

```
myList = list(set(myList)) # Remove duplicates from myList
```

You can transform list data to a set for faster membership look-ups. For example, let's say bigList is a list of the first 10 million integer numbers represented as decimal strings:

```
bigList = [str(i) for i in range(10000000)]
"abc" in bigList # Takes 0.2 sec
bigSet = set(bigList)
"abc" in bigSet # Takes 15–30 μsec—10000 times faster!
```

Dictionaries map keys to values. An object of any hashable data type (number, Boolean, string, tuple) can be a key, and different keys in the same dictionary can belong to different data types. There is no restriction on the data types of dictionary values. Dictionaries have sublinear O(log(N)) search time. They are excellent for key-value look-ups.

You can create a dictionary from a list of (key, value) tuples, and you can use a built-in class constructor enumerate(seq) to create a dictionary where the key is the sequence number of an item in seq:

```
seq = ["alpha", "bravo", "charlie", "delta"]
dict(enumerate(seq))
```

```
⇒ {0: 'alpha', 1: 'bravo', 2: 'charlie', 3: 'delta'}
```

Another smart way to create a dictionary from a sequence of keys (kseq) and a sequence of values (vsec) is through a built-in class constructor, zip(kseq, vseq) (the sequences must be of the same length):

```
kseq = "abcd" # A string is a sequence, too
vseq = ["alpha", "bravo", "charlie", "delta"]
dict(zip(kseq, vseq))
```

```
⇛ {'a': 'alpha', 'c': 'charlie', 'b': 'bravo', 'd': 'delta'}
```

Python implements enumerate(seq) and zip(kseq, vseq) (and the good old range(), too) as list generators. List generators provide an iterator interface, which makes it possible to use them in for loops. Unlike a real list, a list generator produces the next element in a lazy way, only as needed. Generators facilitate working with large lists and even permit “infinite” lists. You can explicitly coerce a generator to a list by calling the list() function.

Unit 6

Comprehending Lists Through List Comprehension

List comprehension is an expression that transforms a collection (not necessarily a list) into a list. It is used to apply the same operation to all or some list elements, such as converting all elements to uppercase or raising them all to a power.

The transformation process looks like this:

1. The expression iterates over the collection and visits the items from the collection.
2. An optional Boolean expression (default True) is evaluated for each item.
3. If the Boolean expression is True, the loop expression is evaluated for the current item, and its value is appended to the result list.
4. If the Boolean expression is False, the item is ignored.

Here are some trivial list comprehensions:

```
# Copy myList; same as myList.copy() or myList[:], but less efficient
[x for x in myList]
# Extract non-negative items
[x for x in myList if x >= 0]
# Build a list of squares
[x**2 for x in myList]
# Build a list of valid reciprocals
[1/x for x in myList if x != 0]
# Collect all non-empty lines from the open file infile,
# with trailing and leading whitespaces removed
[l.strip() for l in infile if l.strip()]
```

In the latter example, the function strip() is evaluated twice for each list item. If you don't want the duplication, you can use nested list comprehensions. The inner one strips off the whitespaces, and the outer one eliminates empty strings:

```
[line for line in [l.strip() for l in infile] if line]
```

If you enclose a list comprehension in parentheses rather than in square brackets, it evaluates to a list generator object:

```
(x**2 for x in myList) # Evaluates to <generator object <genexpr> at 0x...>
```

Often the result of list comprehension is a list of repeating items: numbers, words, word stems, and lemmas. You want to know which item is the most or least common. `Counter` class, coming up next in Unit 7, *Counting with Counters*, on page ?, is a poor man's tool for collecting these sorts of statistics.