

THIRD EDITION

SAMPLE CHAPTER

THE Quick Python Book

Naomi Ceder

Foreword by Nicholas Tollervey



 MANNING



The Quick Python Book
Third Edition
by Naomi Ceder

Sample Chapter 4

Copyright 2018 Manning Publications

brief contents

PART 1 STARTING OUT 1

- 1 ■ About Python 3
- 2 ■ Getting started 11
- 3 ■ The Quick Python overview 20

PART 2 THE ESSENTIALS 35

- 4 ■ The absolute basics 37
- 5 ■ Lists, tuples, and sets 49
- 6 ■ Strings 68
- 7 ■ Dictionaries 89
- 8 ■ Control flow 99
- 9 ■ Functions 113
- 10 ■ Modules and scoping rules 127
- 11 ■ Python programs 142
- 12 ■ Using the filesystem 158
- 13 ■ Reading and writing files 175
- 14 ■ Exceptions 190

PART 3 ADVANCED LANGUAGE FEATURES 205

- 15 ■ Classes and object-oriented programming 207
- 16 ■ Regular expressions 231

17	■	Data types as objects	241
18	■	Packages	255
19	■	Using Python libraries	264

PART 4 WORKING WITH DATA 273

20	■	Basic file wrangling	275
21	■	Processing data files	283
22	■	Data over the network	300
23	■	Saving data	319
24	■	Exploring data	337
		Case study	354

Part 2

The essentials

In the chapters that follow, I show you the essentials of Python. I start from the absolute basics of what makes a Python program and move through Python's built-in data types and control structures, as well as defining functions and using modules.

The last chapter of this part moves on to show you how to write standalone Python programs, manipulate files, handle errors, and use classes.

The absolute basics



This chapter covers

- Indenting and block structuring
- Differentiating comments
- Assigning variables
- Evaluating expressions
- Using common data types
- Getting user input
- Using correct Pythonic style

This chapter describes the absolute basics in Python: how to use assignments and expressions, how to type a number or a string, how to indicate comments in code, and so forth. It starts with a discussion of how Python block structures its code, which differs from every other major language.

4.1 *Indentation and block structuring*

Python differs from most other programming languages because it uses whitespace and indentation to determine block structure (that is, to determine what constitutes

the body of a loop, the else clause of a conditional, and so on). Most languages use braces of some sort to do this. Here is C code that calculates the factorial of 9, leaving the result in the variable `r`:

```
/* This is C code */
int n, r;
n = 9;
r = 1;
while (n > 0) {
    r *= n;
    n--;
}
```

The braces delimit the body of the `while` loop, the code that is executed with each repetition of the loop. The code is usually indented more or less as shown, to make clear what's going on, but it could also be written like this:

```
/* And this is C code with arbitrary indentation */
    int n, r;
        n = 9;
        r = 1;
    while (n > 0) {
r *= n;
n--;
    }
```

The code still would execute correctly, even though it's rather difficult to read.

Here's the Python equivalent:

```
# This is Python code. (Yea!)
n = 9
r = 1
while n > 0:
    r = r * n
    n = n - 1
```



Python also supports
C-style `r *= n`



Python also
supports `n -= 1`

Python doesn't use braces to indicate code structure; instead, the indentation itself is used. The last two lines of the previous code are the body of the `while` loop because they come immediately after the `while` statement and are indented one level further than the `while` statement. If those lines weren't indented, they wouldn't be part of the body of the `while`.

Using indentation to structure code rather than braces may take some getting used to, but there are significant benefits:

- It's impossible to have missing or extra braces. You never need to hunt through your code for the brace near the bottom that matches the one a few lines from the top.
- The visual structure of the code reflects its real structure, which makes it easy to grasp the skeleton of code just by looking at it.

- Python coding styles are mostly uniform. In other words, you're unlikely to go crazy from dealing with someone's idea of aesthetically pleasing code. Everyone's code will look pretty much like yours.

You probably use consistent indentation in your code already, so this won't be a big step for you. If you're using IDLE, it automatically indents lines. You just need to backspace out of levels of indentation when desired. Most programming editors and IDEs—Emacs, VIM, and Eclipse, to name a few—provide this functionality as well. One thing that may trip you up once or twice until you get used to it is the fact that the Python interpreter returns an error message if you have a space (or spaces) preceding the commands you enter at a prompt.

4.2 Differentiating comments

For the most part, anything following a # symbol in a Python file is a comment and is disregarded by the language. The obvious exception is a # in a string, which is just a character of that string:

```
# Assign 5 to x
x = 5
x = 3                # Now x is 3
x = "# This is not a comment"
```

You'll put comments in Python code frequently.

4.3 Variables and assignments

The most commonly used command in Python is assignment, which looks pretty close to what you might've used in other languages. Python code to create a variable called `x` and assign the value 5 to that variable is

```
x = 5
```

In Python, unlike in many other computer languages, neither a variable type declaration nor an end-of-line delimiter is necessary. The line is ended by the end of the line. Variables are created automatically when they're first assigned.

Variables in Python: buckets or labels?

The name *variable* is somewhat misleading in Python; *name* or *label* would be more accurate. However, it seems that pretty much everyone calls variables *variables* at some time or another. Whatever you call them, you should know how they really work in Python.

A common, but inaccurate, explanation is that a variable is a container that stores a value, somewhat like a bucket. This would be reasonable for many programming languages (C, for example).

(continued)

However, in Python variables aren't buckets. Instead, they're labels or tags that refer to objects in the Python interpreter's namespace. Any number of labels (or variables) can refer to the same object, and when that object changes, the value referred to by *all* of those variables also changes.

To see what this means, look at the following simple code:

```
>>> a = [1, 2, 3]
>>> b = a
>>> c = b
>>> b[1] = 5
>>> print(a, b, c)
[1, 5, 3] [1, 5, 3] [1, 5, 3]
```

If you're thinking of variables as containers, this result makes no sense. How could changing the contents of one container simultaneously change the other two? However, if variables are just labels referring to objects, it makes sense that changing the object that all three labels refer to would be reflected everywhere.

If the variables are referring to constants or immutable values, this distinction isn't quite as clear:

```
>>> a = 1
>>> b = a
>>> c = b
>>> b = 5
>>> print(a, b, c)
1 5 1
```

Because the objects they refer to can't change, the behavior of the variables in this case is consistent with either explanation. In fact, in this case, after the third line *a*, *b*, and *c* all refer to the same unchangeable integer object with the value 1. The next line, *b* = 5, makes *b* refer to the integer object 5 but doesn't change the references of *a* or *c*.

Python variables can be set to any object, whereas in C and many other languages, variables can store only the type of value they're declared as. The following is perfectly legal Python code:

```
>>> x = "Hello"
>>> print(x)
Hello
>>> x = 5
>>> print(x)
5
```

x starts out referring to the string object "Hello" and then refers to the integer object 5. Of course, this feature can be abused, because arbitrarily assigning the same variable name to refer successively to different data types can make code confusing to understand.

A new assignment overrides any previous assignments. The `del` statement deletes the variable. Trying to print the variable's contents after deleting it results in an error, as though the variable had never been created in the first place:

```
>>> x = 5
>>> print(x)
5
>>> del x
>>> print(x)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'x' is not defined
>>>
```

Here, you have your first look at a *traceback*, which is printed when an error, called an *exception*, has been detected. The last line tells you what exception was detected, which in this case is a `NameError` exception on `x`. After its deletion, `x` is no longer a valid variable name. In this example, the trace returns only line 1, in `<module>` because only the single line has been sent in the interactive mode. In general, the full dynamic call structure of the existing function at the time of the error's occurrence is returned. If you're using IDLE, you obtain the same information with some small differences. The code may look something like this:

```
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    print(x)
NameError: name 'x' is not defined
```

Chapter 14 describes this mechanism in more detail. A full list of the possible exceptions and what causes them is in the Python standard library documentation. Use the index to find any specific exception (such as `NameError`) you receive.

Variable names are case-sensitive and can include any alphanumeric character as well as underscores but must start with a letter or underscore. See section 4.10 for more guidance on the Pythonic style for creating variable names.

4.4 Expressions

Python supports arithmetic and similar expressions; these expressions will be familiar to most readers. The following code calculates the average of 3 and 5, leaving the result in the variable `z`:

```
x = 3
y = 5
z = (x + y) / 2
```

Note that arithmetic operators involving only integers do *not* always return an integer. Even though all the values are integers, division (starting with Python 3) returns a floating-point number, so the fractional part isn't truncated. If you want traditional integer division returning a truncated integer, you can use `//` instead.

Standard rules of arithmetic precedence apply. If you'd left out the parentheses in the last line, the code would've been calculated as $x + (y / 2)$.

Expressions don't have to involve just numerical values; strings, Boolean values, and many other types of objects can be used in expressions in various ways. I discuss these objects in more detail as they're used.

TRY THIS: VARIABLES AND EXPRESSIONS In the Python shell, create some variables. What happens when you try to put spaces, dashes, or other nonalphanumeric characters in the variable name? Play around with a few complex expressions, such as $x = 2 + 4 * 5 - 6 / 3$. Use parentheses to group the numbers in different ways and see how the result changes compared with the original ungrouped expression.

4.5 *Strings*

You've already seen that Python, like most other programming languages, indicates strings through the use of double quotes. This line leaves the string "Hello, World" in the variable `x`:

```
x = "Hello, World"
```

Backslashes can be used to escape characters, to give them special meanings. `\n` means the newline character, `\t` means the tab character, `\\` means a single normal backslash character, and `\"` is a plain double-quote character. It doesn't end the string:

```
x = "\tThis string starts with a \"tab\"."
x = "This string contains a single backslash(\\)."
```

You can use single quotes instead of double quotes. The following two lines do the same thing:

```
x = "Hello, World"
x = 'Hello, World'
```

The only difference is that you don't need to backslash `"` characters in single-quoted strings or `'` characters in double-quoted strings:

```
x = "Don't need a backslash"
x = 'Can\'t get by without a backslash'
x = "Backslash your \" character!"
x = 'You can leave the " alone'
```

You can't split a normal string across lines. This code won't work:

```
# This Python code will cause an ERROR -- you can't split the string
across two lines.
x = "This is a misguided attempt to
put a newline into a string without using backslash-n"
```

But Python offers triple-quoted strings, which let you do this and include single and double quotes without backslashes:

```
x = """Starting and ending a string with triple " characters
permits embedded newlines, and the use of " and ' without
backslashes"""
```

Now `x` is the entire sentence between the `"""` delimiters. (You can use triple single quotes—`' '`—instead of triple double quotes to do the same thing.)

Python offers enough string-related functionality that chapter 6 is devoted to the topic.

4.6 Numbers

Because you're probably familiar with standard numeric operations from other languages, this book doesn't contain a separate chapter describing Python's numeric abilities. This section describes the unique features of Python numbers, and the Python documentation lists the available functions.

Python offers four kinds of numbers: *integers*, *floats*, *complex numbers*, and *Booleans*. An integer constant is written as an integer—0, -11, +33, 123456—and has unlimited range, restricted only by the resources of your machine. A float can be written with a decimal point or in scientific notation: 3.14, -2E-8, 2.718281828. The precision of these values is governed by the underlying machine but is typically equal to double (64-bit) types in C. Complex numbers are probably of limited interest and are discussed separately later in the section. Booleans are either `True` or `False` and behave identically to 1 and 0 except for their string representations.

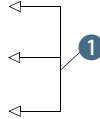
Arithmetic is much like it is in C. Operations involving two integers produce an integer, except for division (`/`), which results in a float. If the `//` division symbol is used, the result is an integer, with truncation. Operations involving a float always produce a float. Here are a few examples:

```
>>> 5 + 2 - 3 * 2
1
>>> 5 / 2          # floating-point result with normal division
2.5
>>> 5 / 2.0        # also a floating-point result
2.5
>>> 5 // 2         # integer result with truncation when divided using '//'
2
>>> 30000000000    # This would be too large to be an int in many languages
30000000000
>>> 30000000000 * 3
90000000000
>>> 30000000000 * 3.0
90000000000.0
>>> 2.0e-8         # Scientific notation gives back a float
2e-08
>>> 3000000 * 3000000
```

```

9000000000000000
>>> int(200.2)
200
>>> int(2e2)
200
>>> float(200)
200.0

```



These are explicit conversions between types ❶. `int` truncates float values.

Numbers in Python have two advantages over C or Java: Integers can be arbitrarily large, and the division of two integers results in a float.

4.6.1 *Built-in numeric functions*

Python provides the following number-related functions as part of its core:

```
abs, divmod, float, hex, int, max, min, oct,
pow, round
```

See the documentation for details.

4.6.2 *Advanced numeric functions*

More advanced numeric functions such as the trig and hyperbolic trig functions, as well as a few useful constants, aren't built into Python but are provided in a standard module called `math`. I explain modules in detail later. For now, it's sufficient to know that you must make the `math` functions in this section available by starting your Python program or interactive session with the statement

```
from math import *
```

The `math` module provides the following functions and constants:

```
acos, asin, atan, atan2, ceil, cos, cosh, e, exp, fabs, floor, fmod,
frexp, hypot, ldexp, log, log10, mod, pi, pow, sin, sinh, sqrt, tan,
tanh
```

See the documentation for details.

4.6.3 *Numeric computation*

The core Python installation isn't well suited to intensive numeric computation because of speed constraints. But the powerful Python extension NumPy provides highly efficient implementations of many advanced numeric operations. The emphasis is on array operations, including multidimensional matrices and more advanced functions such as the Fast Fourier Transform. You should be able to find NumPy (or links to it) at www.scipy.org.

4.6.4 *Complex numbers*

Complex numbers are created automatically whenever an expression of the form `nj` is encountered, with `n` having the same form as a Python integer or float. `j` is, of course,

standard notation for the imaginary number equal to the square root of -1 , for example:

```
>>> (3+2j)
(3+2j)
```

Note that Python expresses the resulting complex number in parentheses as a way of indicating that what's printed to the screen represents the value of a single object:

```
>>> 3 + 2j - (4+4j)
(-1-2j)
>>> (1+2j) * (3+4j)
(-5+10j)
>>> 1j * 1j
(-1+0j)
```

Calculating $j * j$ gives the expected answer of -1 , but the result remains a Python complex-number object. Complex numbers are never converted automatically to equivalent real or integer objects. But you can easily access their real and imaginary parts with `real` and `imag`:

```
>>> z = (3+5j)
>>> z.real
3.0
>>> z.imag
5.0
```

Note that real and imaginary parts of a complex number are always returned as floating-point numbers.

4.6.5 Advanced complex-number functions

The functions in the `math` module don't apply to complex numbers; the rationale is that most users want the square root of -1 to generate an error, not an answer! Instead, similar functions, which can operate on complex numbers, are provided in the `cmath` module:

```
acos, acosh, asin, asinh, atan, atanh, cos, cosh, e, exp, log, log10,
pi, sin, sinh, sqrt, tan, tanh.
```

To make clear in the code that these functions are special-purpose complex-number functions and to avoid name conflicts with the more normal equivalents, it's best to import the `cmath` module by saying

```
import cmath
```

and then to explicitly refer to the `cmath` package when using the function:

```
>>> import cmath
>>> cmath.sqrt(-1)
1j
```

Minimizing from <module> import *

This is a good example of why it's best to minimize the use of the `from <module> import *` form of the `import` statement. If you used it to import first the `math` module and then the `cmath` module, the commonly named functions in `cmath` would override those of `math`. It's also more work for someone reading your code to figure out the source of the specific functions you use. Some modules are explicitly designed to use this form of import.

See chapter 10 for more details on how to use modules and module names.

The important thing to keep in mind is that by importing the `cmath` module, you can do almost anything you can do with other numbers.

TRY THIS: MANIPULATING STRINGS AND NUMBERS In the Python shell, create some string and number variables (integers, floats, *and* complex numbers). Experiment a bit with what happens when you do operations with them, including across types. Can you multiply a string by an integer, for example, or can you multiply it by a float or complex number? Also load the `math` module and try a few of the functions; then load the `cmath` module and do the same. What happens if you try to use one of those functions on an integer or float after loading the `cmath` module? How might you get the `math` module functions back?

4.7 The None value

In addition to standard types such as strings and numbers, Python has a special basic data type that defines a single special data object called `None`. As the name suggests, `None` is used to represent an empty value. It appears in various guises throughout Python. For example, a procedure in Python is just a function that doesn't explicitly return a value, which means that by default, it returns `None`.

`None` is often useful in day-to-day Python programming as a placeholder to indicate a point in a data structure where meaningful data will eventually be found, even though that data hasn't yet been calculated. You can easily test for the presence of `None` because there's only one instance of `None` in the entire Python system (all references to `None` point to the same object), and `None` is equivalent only to itself.

4.8 Getting input from the user

You can also use the `input()` function to get input from the user. Use the prompt string you want to display to the user as `input`'s parameter:

```
>>> name = input("Name? ")
Name? Jane
>>> print(name)
Jane
>>> age = int(input("Age? "))
Age? 28
```

← Converts input
from string to int


```
>>> print(age)
28
>>>
```

This is a fairly simple way to get user input. The one catch is that the input comes in as a string, so if you want to use it as a number, you have to use the `int()` or `float()` function to convert it.

TRY THIS: GETTING INPUT Experiment with the `input()` function to get string and integer input. Using code similar to the previous code, what is the effect of not using `int()` around the call to `input()` for integer input? Can you modify that code to accept a float—say, 28.5? What happens if you deliberately enter the wrong type of value? Examples include a float in which an integer is expected and a string in which a number is expected—and vice versa.

4.9 Built-in operators

Python provides various built-in operators, from the standard (+, *, and so on) to the more esoteric, such as operators for performing bit shifting, bitwise logical functions, and so forth. Most of these operators are no more unique to Python than to any other language; hence, I won't explain them in the main text. You can find a complete list of the Python built-in operators in the documentation.

4.10 Basic Python style

Python has relatively few limitations on coding style with the obvious exception of the requirement to use indentation to organize code into blocks. Even in that case, the amount of indentation and type of indentation (tabs versus spaces) isn't mandated. However, there are preferred stylistic conventions for Python, which are contained in Python Enhancement Proposal (PEP) 8, which is summarized in appendix A and available online at www.python.org/dev/peps/pep-0008/. A selection of Pythonic conventions is provided in table 4.1, but to fully absorb Pythonic style, periodically reread PEP 8.

Table 4.1 Pythonic coding conventions

Situation	Suggestion	Example
Module/package names	Short, all lowercase, underscores only if needed	<code>imp, sys</code>
Function names	All lowercase, underscores_for_readablitiy	<code>foo(), my_func()</code>
Variable names	All lowercase, underscores_for_readablitiy	<code>my_var</code>
Class names	CapitalizeEachWord	<code>MyClass</code>
Constant names	ALL_CAPS_WITH_UNDERSCORES	<code>PI, TAX_RATE</code>

Table 4.1 Pythonic coding conventions (*continued*)

Situation	Suggestion	Example
Indentation	Four spaces per level, no tabs	
Comparisons	Don't compare explicitly to True or False	<pre>if my_var: if not my_var:</pre>

I strongly urge you to follow the conventions of PEP 8. They're wisely chosen and time-tested, and they'll make your code easier for you and other Python programmers to understand.

QUICK CHECK: PYTHONIC STYLE Which of the following variable and function names do you think are not good Pythonic style? Why?

```
bar(), varName, VERYLONGVARNAME, foobar, longvarname,
foo_bar(), really_very_long_var_name
```

Summary

- The basic syntax summarized above is enough to start writing Python code.
- Python syntax is predictable and consistent.
- Because the syntax offers few surprises, many programmers can get started writing code surprisingly quickly.

The Quick Python Book Third Edition

Naomi Ceder

Initially Guido van Rossum's 1989 holiday project, Python has grown into an amazing computer language. It's a joy to learn and read, and powerful enough to handle everything from low-level system resources to advanced applications like deep learning. Elegantly simple and complete, it also boasts a massive ecosystem of libraries and frameworks. Python programmers are in high demand—you can't afford not to be fluent!

The Quick Python Book, Third Edition is a comprehensive guide to the Python language by a Python authority, Naomi Ceder. With the personal touch of a skilled teacher, she beautifully balances details of the language with the insights and advice you need to handle any task. Extensive, relevant examples and learn-by-doing exercises help you master each important concept the first time through. Whether you're scraping websites or playing around with nested tuples, you'll appreciate this book's clarity, focus, and attention to detail.

What's Inside

- Clear coverage of Python 3
- Core libraries, packages, and tools
- In-depth exercises
- Five new data science–related chapters

Written for readers familiar with programming concepts—no Python experience assumed.

Naomi Ceder is chair of the Python Software Foundation. She has been learning, using, and teaching Python since 2001.

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit
manning.com/books/the-quick-python-book-third-edition

“Naomi’s book epitomizes what it is to be Pythonic: beautiful is better than ugly, simple is better than complex, and readability counts.”

—From the Foreword by
 Nicholas Tollervey
 Python Software Foundation

“Leads you from a basic knowledge of Python to its most interesting features, always using accessible language.”

—Eros Pedrini, everis

“Unleash your Pythonic powers with this book and start coding real-world applications fast.”

—Carlos Fernández Manzano
 Aguas de Murcia

“The complete and definitive book to start learning Python.”

—Christos Paisios, e-Travel



ISBN-13: 978-1-61729-403-7
 ISBN-10: 1-61729-403-9



9 781617 294037



\$39.99 / Can \$52.99 [INCLUDING eBook]