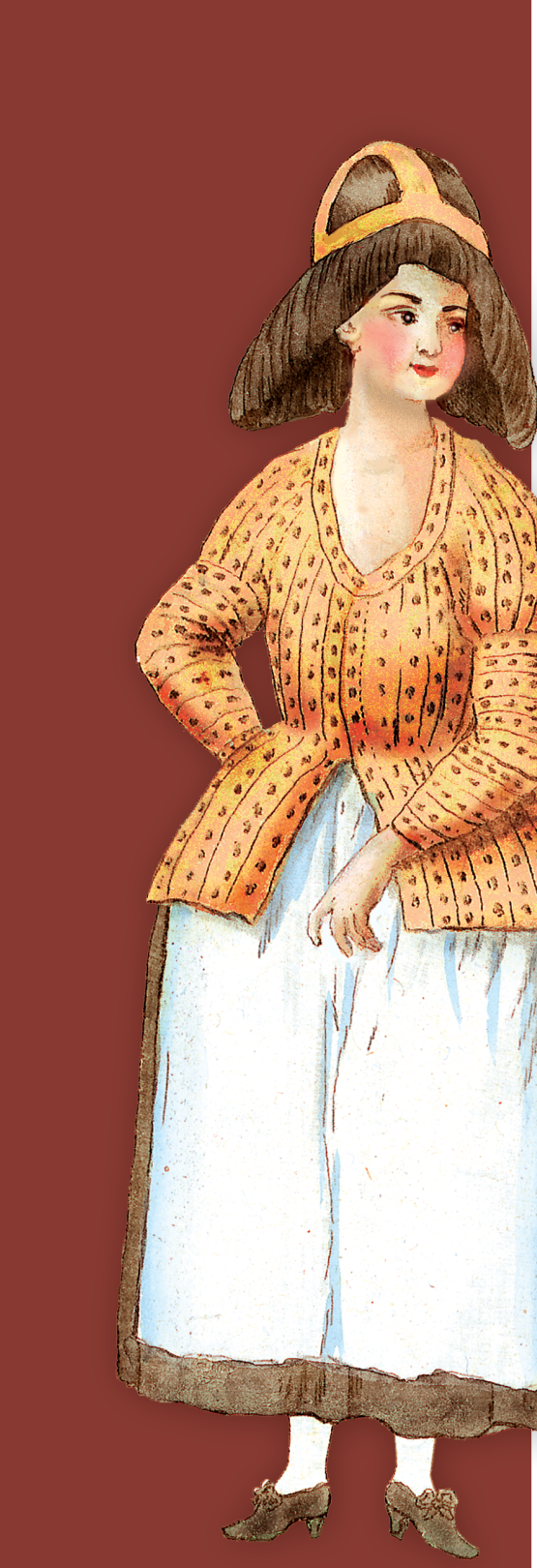

SECOND EDITION Covers Python 3

SAMPLE CHAPTER

THE Quick Python Book

First edition by Daryl K. Harms
Kenneth M. McDonald

Naomi R. Ceder

MANNING

The Quick Python Book
Second Edition

by Naomi R. Ceder

Chapter 4

brief contents

Part 2

The essentials

In the chapters that follow, we'll show you the essentials of Python. We'll 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 section of this part moves on to show you how to write standalone Python programs, manipulate files, handle errors, and use classes. The section ends with chapter 16, which is a brief introduction to GUI programming using Python's `tkinter` module.

4 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: assignments and expressions, how to type a number or a string, how to indicate comments in code, and so forth. It starts out with a discussion of how Python block structures its code, which is different from any 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 `{` and `}` 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--;
}
```

It 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 they 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'll 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. This 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. Their 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, including 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 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"
```

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

Python variables can be set to any object, unlike C or many other languages' variables, which 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 gives an error the same as if 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 we have our first look at a *traceback*, which is printed when an error, called an *exception*, has been detected. The last line tells us 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 calls at the time of the occurrence of the error is returned. If you're using IDLE, you obtain the same information with some small differences; it 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 will describe this mechanism in more detail. A full list of the possible exceptions and what causes them is in the appendix of this book. 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 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 unlike the arithmetic rules of C in terms of type coercions, 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 the `//` instead.

Standard rules of arithmetic precedence apply; if we'd left out the parentheses in the last line, it 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. We'll discuss these in more detail as they're used.

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 permit single and double quotes to be included 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 also 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 using 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 (`/`), where a float results. 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
9000000000000
>>> int(200.2)          (1)
200
>>> int(2e2)            (1)
200
>>> float(200)          (1)
200.0
```

These are explicit conversions between types (1). `int` will truncate float values.

Numbers in Python have two advantages over C or Java. First, integers can be arbitrarily large; and second, 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, long, 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-ins in Python but are provided in a standard module called `math`. Modules will be explained in detail later; for now, it's sufficient to know that the math functions in this section must be made 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, acosh, asin, asinh, atan, atan2, atanh, ceil, copysign,
cos, cosh, degrees, e, exp, fabs, factorial, floor, fmod,
frexp, fsum, hypot, isinf, isnan, ldexp, log, log10, log1p,
modf, pi, pow, radians, sin, sinh, sqrt, tan, tanh, trunc
```

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 engineering 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 is 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,
isinf, isnan, log, log10, phase, pi, polar, rect, sin, sinh,
sqrt, tan, tanh
```

In order to make clear in the code that these 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 imported first the `math` and then the `cmath` modules using it, 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.

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 is 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 displayed to the user as `input`'s parameter:

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

Converts input from string to int

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.

4.9 Built-in operators

Python provides various built-in operators, from the standard (such as +, *, 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, and 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 the appendix and can be found 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 you'll need to periodically reread PEP 8.

Table 4.1 Pythonic coding conventions

Situation	Suggestion	Example
Module/package names	short, all lowercase, underscores only if needed	`imp, sys`
Function names	all lowercase, underscores_for_readablitiy	`foo(), my_func()`
Variable names	all lowercase, underscores_for_readablitiy	`my_var`
Class names	CapitalizeEachWord	`MyClass`
Constant names	ALL_CAPS_WITH_UNDERSCORES	`PI, TAX_RATE`
Indentation	4 spaces per level, don't use tabs	
Comparisons	Don't compare explicitly to True or False	`if my_var:` `if not my_var:`

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

4.11 Summary

That's the view of Python from 30,000 feet. If you're an experienced programmer, you're probably already seeing how you can write your code in Python. If that's the case, you should feel free to start experimenting with your own code. Many programmers find it surprisingly easy to pick up Python syntax, because there are relatively few surprises. Once you pick up the basics of the language, it's very predictable and consistent.

In any case, we have just covered the broadest outlines of the language, and there are lots of details that we still need to cover, beginning in the next chapter with one of the workhorses of Python, lists.

PROGRAMMING

THE Quick Python Book SECOND EDITION

Naomi R. Ceder

This revision of Manning's popular **The Quick Python Book** offers a clear, crisp introduction to the elegant Python programming language and its famously easy-to-read syntax. Written for programmers new to Python, this updated edition covers features common to other languages concisely, while introducing Python's comprehensive standard functions library and unique features in detail.

After exploring Python's syntax, control flow, and basic data structures, the book shows how to create, test, and deploy full applications and larger code libraries. It addresses established Python features as well as the advanced object-oriented options available in Python 3. Along the way, you'll survey the current Python development landscape, including GUI programming, testing, database access, and web frameworks.

What's Inside

- Concepts and Python 3 features
- Regular expressions and testing
- Python tools
- All the Python you need—nothing you don't

Second edition author **Naomi Ceder** is IT Director at Zoro Tools, Inc., in Buffalo Grove, Illinois, and an organizer and teacher of the Chicago Python Workshops. The first edition of this book was written by **Daryl Harms** and **Kenneth McDonald**.

For online access to the author, and a free ebook for owners of this book, go to manning.com/TheQuickPythonBookSecondEdition

"The quickest way to learn the basics of Python."
—Massimo Perga, Microsoft

"This is my favorite Python book... a competent way into serious Python programming."
—Edmon Begoli
Oak Ridge National Laboratory

"Great book... covers the new incarnation of Python."
—William Kahn-Greene
Participatory Culture Foundation

"Like Python itself, its emphasis is on readability and rapid development."
—David McWhirter, Cranberryink

"Python coders will love this nifty book."
—Sumit Pal, Leapfrogrx

ISBN 13: 978-1-935182-20-7
ISBN 10: 1-935182-20-X
53999
9 781935 182207

MANNING $39.99 / Can $49.99 [INCLUDING eBOOK]