

SECOND EDITION

Covers Python 3

Sample Chapter

THE Quick Python Book

First edition by Daryl K. Harms
Kenneth M. McDonald

Vernon L. Ceder



 MANNING



The Quick Python Book
Second Edition
by Vernon L. Ceder

Chapter 8

Copyright 2010 Manning Publications

brief contents

PART 1 STARTING OUT..... 1

- 1 ■ About Python 3
- 2 ■ Getting started 10
- 3 ■ The Quick Python overview 18

PART 2 THE ESSENTIALS..... 33

- 4 ■ The absolute basics 35
- 5 ■ Lists, tuples, and sets 45
- 6 ■ Strings 63
- 7 ■ Dictionaries 81
- 8 ■ Control flow 90
- 9 ■ Functions 103
- 10 ■ Modules and scoping rules 115
- 11 ■ Python programs 129
- 12 ■ Using the filesystem 147
- 13 ■ Reading and writing files 159
- 14 ■ Exceptions 172
- 15 ■ Classes and object-oriented programming 186
- 16 ■ Graphical user interfaces 209

PART 3 ADVANCED LANGUAGE FEATURES 223

- 17 ■ Regular expressions 225
- 18 ■ Packages 234
- 19 ■ Data types as objects 242
- 20 ■ Advanced object-oriented features 247

PART 4 WHERE CAN YOU GO FROM HERE? 263

- 21 ■ Testing your code made easy(-er) 265
- 22 ■ Moving from Python 2 to Python 3 274
- 23 ■ Using Python libraries 282
- 24 ■ Network, web, and database programming 290

Control flow

This chapter covers

- Repeating code with a `while` loop
- Making decisions: the `if-elif-else` statement
- Iterating over a list with a `for` loop
- Using list and dictionary comprehensions
- Delimiting statements and blocks with indentation
- Evaluating Boolean values and expressions

Python provides a complete set of control-flow elements, with loops and conditionals. This chapter examines each in detail.

8.1 The while loop

You've come across the basic `while` loop several times already. The full `while` loop looks like this:

```
while condition:
    body
else:
    post-code
```

condition is an expression that evaluates to a true or false value. As long as it's True, the body will be executed repeatedly. If it evaluates to False, the while loop will execute the post-code section and then terminate. If the condition starts out by being false, the body won't be executed at all—just the post-code section. The body and post-code are each sequences of one or more Python statements that are separated by newlines and are at the same level of indentation. The Python interpreter uses this level to delimit them. No other delimiters, such as braces or brackets, are necessary.

Note that the else part of the while loop is optional and not often used. That's because as long as there is no break in the body, this loop

```
while condition:
    body
else:
    post-code
```

and this loop

```
while condition:
    body
post-code
```

do the same things—and the second is simpler to understand. I probably wouldn't have mentioned the else clause except that if you haven't learned about it by now, you may have found it confusing if you found this syntax in another person's code. Also, it's useful in some situations.

8.1.1 *The break and continue statements*

The two special statements break and continue can be used in the body of a while loop. If break is executed, it immediately terminates the while loop, and not even the post-code (if there is an else clause) will be executed. If continue is executed, it causes the remainder of the body to be skipped over; the condition is evaluated again, and the loop proceeds as normal.

8.2 *The if-elif-else statement*

The most general form of the if-then-else construct in Python is

```
if condition1:
    body1
elif condition2:
    body2
elif condition3:
    body3
.
.
.
elif condition(n-1):
    body(n-1)
```

```
else:
    body(n)
```

It says: if `condition1` is true, execute `body1`; otherwise, if `condition2` is true, execute `body2`; otherwise ... and so on, until it either finds a condition that evaluates to True or hits the `else` clause, in which case it executes `body(n)`. As for the `while` loop, the `body` sections are again sequences of one or more Python statements that are separated by newlines and are at the same level of indentation.

Of course, you don't need all that luggage for every conditional. You can leave out the `elif` parts, or the `else` part, or both. If a conditional can't find any body to execute (no conditions evaluate to True, and there is no `else` part), it does nothing.

The body after the `if` statement is required. But you can use the `pass` statement here (as you can anywhere in Python where a statement is required). The `pass` statement serves as a placeholder where a statement is needed, but it performs no action:

```
if x < 5:
    pass
else:
    x = 5
```

There is no case (or switch) statement in Python.

8.3 *The for loop*

A `for` loop in Python is different from `for` loops in some other languages. The traditional pattern is to increment and test a variable on each iteration, which is what C `for` loops usually do. In Python, a `for` loop iterates over the values returned by any iterable object—that is, any object that can yield a sequence of values. For example, a `for` loop can iterate over every element in a list, a tuple, or a string. But an iterable object can also be a special function called `range` or a special type of function called a *generator*. This can be quite powerful. The general form is

```
for item in sequence:
    body
else:
    post-code
```

`body` will be executed once for each element of `sequence`. `variable` is set to be the first element of `sequence`, and `body` is executed; then, `variable` is set to be the second element of `sequence`, and `body` is executed; and so on, for each remaining element of the `sequence`.

The `else` part is optional. As with the `else` part of a `while` loop, it's rarely used. `break` and `continue` do the same thing in a `for` loop as in a `while` loop.

This small loop prints out the reciprocal of each number in `x`:

```
x = [1.0, 2.0, 3.0]
for n in x:
    print(1 / n)
```

8.3.1 The range function

Sometimes you need to loop with explicit indices (to use the position at which values occur in a list). You can use the range command together with the len command on lists to generate a sequence of indices for use by the for loop. This code prints out all the positions in a list where it finds negative numbers:

```
x = [1, 3, -7, 4, 9, -5, 4]
for i in range(len(x)):
    if x[i] < 0:
        print("Found a negative number at index ", i)
```

Given a number n , range(n) returns a sequence 0, 1, 2, ..., $n-2$, $n-1$. So, passing it the length of a list (found using len) produces a sequence of the indices for that list's elements. The range function doesn't build a Python list of integers—it just appears to. Instead, it creates a range object that produces integers on demand. This is useful when you're using explicit loops to iterate over really large lists. Instead of building a list with 10 million elements in it, for example, which would take up quite a bit of memory, you can use range(10000000), which takes up only a small amount of memory and generates a sequence of integers from 0 up to 10000000 as needed by the for loop.

CONTROLLING STARTING AND STEPPING VALUES WITH RANGE

You can use two variants on the range function to gain more control over the sequence it produces. If you use range with two numeric arguments, the first argument is the starting number for the resulting sequence and the second number is the number the resulting sequence goes up to (but doesn't include). Here are a few examples:

```
>>> list(range(3, 7))
[3, 4, 5, 6]
>>> list(range(2, 10))
[2, 3, 4, 5, 6, 7, 8, 9]
>>> list(range(5, 3))
[]
```

list() is used only to force the items range would generate to appear as a list. It's not normally used in actual code ❶.

This still doesn't allow you to count backward, which is why the value of range(5, 3) is an empty list. To count backward, or to count by any amount other than 1, you need to use the optional third argument to range, which gives a step value by which counting proceeds:

```
>>> list(range(0, 10, 2))
[0, 2, 4, 6, 8]
>>> list(range(5, 0, -1))
[5, 4, 3, 2, 1]
```

Sequences returned by range always include the starting value given as an argument to range and never include the ending value given as an argument.

8.3.2 *Using break and continue in for loops*

The two special statements `break` and `continue` can also be used in the body of a `for` loop. If `break` is executed, it immediately terminates the `for` loop, and not even the post-code (if there is an `else` clause) will be executed. If `continue` is executed in a `for` loop, it causes the remainder of the body to be skipped over, and the loop proceeds as normal with the next item. .

8.3.3 *The for loop and tuple unpacking*

You can use tuple unpacking to make some `for` loops cleaner. The following code takes a list of two-element tuples and calculates the value of the sum of the products of the two numbers in each tuple (a moderately common mathematical operation in some fields):

```
somelist = [(1, 2), (3, 7), (9, 5)]
result = 0
for t in somelist:
    result = result + (t[0] * t[1])
```

Here's the same thing, but cleaner:

```
somelist = [(1, 2), (3, 7), (9, 5)]
result = 0

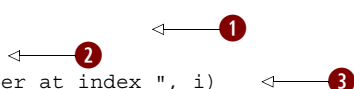
for x, y in somelist:
    result = result + (x * y)
```

We use a tuple `x, y` immediately after the `for` keyword, instead of the usual single variable. On each iteration of the `for` loop, `x` contains element 0 of the current tuple from `list`, and `y` contains element 1 of the current tuple from `list`. Using a tuple in this manner is a convenience of Python, and doing this indicates to Python that each element of the list is expected to be a tuple of appropriate size to unpack into the variable names mentioned in the tuple after the `for`.

8.3.4 *The enumerate function*

You can combine tuple unpacking with the `enumerate` function to loop over both the items and their index. This is similar to using `range` but has the advantage that the code is clearer and easier to understand. Like the previous example, the following code prints out all the positions in a list where it finds negative numbers:

```
x = [1, 3, -7, 4, 9, -5, 4]
for i, n in enumerate(x):
    if n < 0:
        print("Found a negative number at index ", i)
```



The `enumerate` function returns tuples of (index, item) **1**. You can access the item without the index **2**. The index is also available **3**.

8.3.5 The zip function

Sometimes it's useful to combine two or more iterables before looping over them. The `zip` function will take the corresponding elements from one or more iterables and combine them into tuples until it reaches the end of the shortest iterable:

```
>>> x = [1, 2, 3, 4]
>>> y = ['a', 'b', 'c']
>>> z = zip(x, y)
>>> list(z)
[(1, 'a'), (2, 'b'), (3, 'c')]
```

← y is 3 elements;
x is 4 elements

← z has only 3
elements

8.4 List and dictionary comprehensions

The pattern of using a `for` loop to iterate through a list, modify or select individual elements, and create a new list or dictionary is very common. Such loops often look a lot like the following:

```
>>> x = [1, 2, 3, 4]
>>> x_squared = []
>>> for item in x:
...     x_squared.append(item * item)
...
>>> x_squared
[1, 4, 9, 16]
```

This sort of situation is so common that Python has a special shortcut for such operations, called a *comprehension*. You can think of a list or dictionary comprehension as a one-line `for` loop that creates a new list or dictionary from another list. The pattern of a list comprehension is as follows:

```
new_list = [expression for variable in old_list if expression]
```

And a dictionary comprehension looks like this:

```
new_dict = {expression:expression for variable in list if expression}
```

In both cases, the heart of the expression is similar to the beginning of a `for` loop—`for variable in list`—with some expression using that variable to create a new key or value and an optional conditional expression using the value of the variable to select whether it's included in the new list or dictionary. For example, the following code does exactly the same thing as the previous code but is a list comprehension:

```
>>> x = [1, 2, 3, 4]
>>> x_squared = [item * item for item in x]
>>> x_squared
[1, 4, 9, 16]
```

You can even use `if` statements to select items from the list:

```
>>> x = [1, 2, 3, 4]
>>> x_squared = [item * item for item in x if item > 2]
```

```
>>> x_squared
[9, 16]
```

Dictionary comprehensions are similar, but you need to supply both a key and a value. If we want to do something similar to the previous example but have the number be the key and the number's square be the value in a dictionary, we can use a dictionary comprehension, like so:

```
>>> x = [1, 2, 3, 4]
>>> x_squared_dict = {item: item * item for item in x}
>>> x_squared_dict
{1: 1, 2: 4, 3: 9, 4: 16}
```

List and dictionary comprehensions are very flexible and powerful, and when you get used to them they make list-processing operations much simpler. I recommend that you experiment with them and try them anytime you find yourself writing a for loop to process a list of items.

8.5 **Statements, blocks, and indentation**

Because the control flow constructs we encountered in this chapter are the first to make use of blocks and indentation, this is a good time to revisit the subject.

Python uses the indentation of the statements to determine the delimitation of the different blocks (or bodies) of the control-flow constructs. A block consists of one or more statements, which are usually separated by newlines. Examples of Python statements are the assignment statement, function calls, the print function, the placeholder pass statement, and the del statement. The control-flow constructs (if-elif-else, while, and for loops) are compound statements:

```
compound statement clause:
    block
compound statement clause:
    block
```

A compound statement contains one or more clauses that are each followed by indented blocks. Compound statements can appear in blocks just like any other statements. When they do, they create nested blocks.

You may also encounter a couple of special cases. Multiple statements may be placed on the same line if they are separated by semicolons. A block containing a single line may be placed on the same line after the semicolon of a clause of a compound statement:

```
>>> x = 1; y = 0; z = 0
>>> if x > 0: y = 1; z = 10
... else: y = -1
...
>>> print(x, y, z)
1 1 10
```

Improperly indented code will result in an exception being raised. You may encounter two forms of this. The first is

```
>>>
>>> x = 1
File "<stdin>", line 1
    x = 1
    ^
IndentationError: unexpected indent
>>>
```

We indented a line that should not have been indented. In the basic mode, the carat (^) indicates the spot where the problem occurred. In the IDLE Python Shell (see figure 8.1), the invalid indent is highlighted. The same message would occur if we didn't indent where necessary (that is, the first line after a compound statement clause).

One situation where this can occur can be confusing. If you're using an editor that displays tabs in four-space increments (or the Windows interactive mode, which indents the first tab only four spaces from the prompt) and indent one line with four spaces and then the next line with a tab, the two lines may appear to be at the same level of indentation. But you'll receive this exception because Python maps the tab to eight spaces. The best way to avoid this problem is to use only spaces in Python code. If you must use tabs for indentation, or if you're dealing with code that uses tabs, be sure never to mix them with spaces.

On the subject of the basic interactive mode and the IDLE Python Shell, you likely have noticed that you need an extra line after the outermost level of indentation:

```
>>> x = 1
>>> if x == 1:
...     y = 2
...     if v > 0:
...         z = 2
...         v = 0
...
>>> x = 2
```

No line is necessary after the line `z = 2`, but one is needed after the line `v = 0`. This line is unnecessary if you're placing your code in a module in a file.

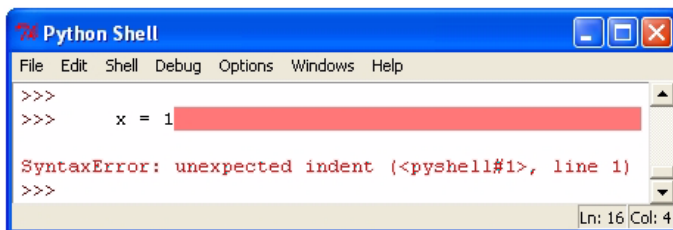


Figure 8.1 Indentation error

The second form of exception will occur if you indent a statement in a block less than the legal amount:

```
>>> x = 1
>>> if x == 1:
    y = 2
    z = 2
File "<stdin>", line 3
    z = 2
    ^
IndentationError: unindent does not match any outer indentation level
```

Here, the line containing `z = 2` isn't lined up properly under the line containing `y = 2`. This form is rare, but I mention it again because in a similar situation, it may be confusing.

Python will allow you to indent any amount and won't complain regardless of how much you vary it as long as you're consistent within a single block. Please don't take improper advantage of this. The recommended standard is to use four spaces for each level of indentation.

Before leaving indentation, I'll cover breaking up statements across multiple lines. This of course is necessary more often as the level of indentation increases. You can explicitly break up a line using the backslash character. You can also implicitly break any statement between tokens when within a set of `()`, `{}`, or `[]` delimiters (that is, when typing a set of values in a list, a tuple, or a dictionary, or a set of arguments in a function call, or any expression within a set of brackets). You can indent the continuation line of a statement to any level you desire:

```
>>> print('string1', 'string2', 'string3' \
...       , 'string4', 'string5')
string1 string2 string3 string4 string5
>>> x = 100 + 200 + 300 \
...     + 400 + 500
>>> x
1500
>>> v = [100, 300, 500, 700, 900,
...       1100, 1300]
>>> v
[100, 300, 500, 700, 900, 1100, 1300]
>>> max(1000, 300, 500,
...      800, 1200)
1200
>>> x = (100 + 200 + 300
...      + 400 + 500)
>>> x
1500
```

You can break a string with a `\` as well. But any indentation tabs or spaces will become part of the string, and the line *must* end with the `\`. To avoid this, you can use the fact that any set of string literals separated by whitespace is automatically concatenated:

```
>>> "strings separated by whitespace " \
...   "'are automatically'" " ' concatenated'
'strings separated by whitespace are automatically concatenated'
>>> x = 1
>>> if x > 0:
...     string1 = "this string broken by a backslash will end up \
...               with the indentation tabs in it"
...
>>> string1
'this string broken by a backslash will end up \t\t\twith
  the indentation tabs in it'
>>> if x > 0:
...     string1 = "this can be easily avoided by splitting the " \
...               "string in this way"
...
>>> string1
'this can be easily avoided by splitting the string in this way'
```

8.6 Boolean values and expressions

The previous examples of control flow use conditional tests in a fairly obvious manner but never really explain what constitutes true or false in Python or what expressions can be used where a conditional test is needed. This section describes these aspects of Python.

Python has a Boolean object type that can be set to either True or False. Any expression with a Boolean operation will return True or False.

8.6.1 Most Python objects can be used as Booleans

In addition, Python is similar to C with respect to Boolean values, in that C uses the integer 0 to mean false and any other integer to mean true. Python generalizes this idea; 0 or empty values are False, and any other values are True. In practical terms, this means the following:

- The numbers 0, 0.0, and 0+0j are all False; any other number is True.
- The empty string "" is False; any other string is True.
- The empty list [] is False; any other list is True.
- The empty dictionary {} is False; any other dictionary is True.
- The empty set set () is False; any other set is True.
- The special Python value None is always False.

There are some Python data structures we haven't looked at yet, but generally the same rule applies. If the data structure is empty or 0, it's taken to mean false in a Boolean context; otherwise it's taken to mean true. Some objects, such as file objects and code objects, don't have a sensible definition of a 0 or empty element, and these objects shouldn't be used in a Boolean context.

8.6.2 *Comparison and Boolean operators*

You can compare objects using normal operators: `<`, `<=`, `>`, `>=`, and so forth. `==` is the equality test operator, and either `!=` or `<>` may be used as the “not equal to” test. There are also `in` and `not in` operators to test membership in sequences (lists, tuples, strings, and dictionaries) as well as `is` and `is not` operators to test whether two objects are the same.

Expressions that return a Boolean value may be combined into more complex expressions using the `and`, `or`, and `not` operators. This code snippet checks to see if a variable is within a certain range:

```
if 0 < x and x < 10:
    ...
```

Python offers a nice shorthand for this particular type of compound statement; you can write it as you would in a math paper:

```
if 0 < x < 10:
    ...
```

Various rules of precedence apply; when in doubt, you can use parentheses to make sure Python interprets an expression the way you want it to. This is probably a good idea for complex expressions, regardless of whether it's necessary, because it makes it clear to future maintainers of the code exactly what's happening. See the appendix for more details on precedence.

The rest of this section provides more advanced information. If this is your first read through this book as you're learning the language, you may want to skip over it.

The `and` and `or` operators return objects. The `and` operator returns either the first false object (that an expression evaluates to) or the last object. Similarly, the `or` operator returns either the first true object or the last object. As with many other languages, evaluation stops as soon as a true expression is found for the `or` operator or as soon as a false expression is found for the `and` operator:

```
>>> [2] and [3, 4]
[3, 4]
>>> [] and 5
[]
>>> [2] or [3, 4]
[2]
>>> [] or 5
5
>>>
```

The `==` and `!=` operators test to see if their operands contains the same values. They are used in most situations. The `is` and `is not` operators test to see if their operands are the same object:

```
>>> x = [0]
>>> y = [x, 1]
>>> x is y[0]
True
>>> x = [0]
>>> x is y[0]
False
>>> x == y[0]
True
```

They reference the same object

x has been assigned to a different object

Revisit “Nested lists and deep copies” (section 5.6) of “Lists, tuples, and sets” (chapter 5) if this example isn’t clear to you.

8.7 Writing a simple program to analyze a text file

To give you a better sense of how a Python program works, let’s look a small sample that roughly replicates the UNIX `wc` utility and reports the number of lines, words, and characters in a file. The sample in listing 8.1 is deliberately written to be clear to programmers new to Python and as simple as possible.

Listing 8.1 `word_count.py`

```
#!/usr/bin/env python3.1

""" Reads a file and returns the number of lines, words,
    and characters - similar to the UNIX wc utility
"""

infile = open('word_count.tst')
lines = infile.read().split("\n")

line_count = len(lines)

word_count = 0
char_count = 0

for line in lines:
    words = line.split()
    word_count += len(words)

    char_count += len(line)

print("File has {0} lines, {1} words, {2} characters".format
      count, word_count, char_count))
```

Opens file

Reads file; splits into lines

Gets number of lines with len()

Initializes other counts

Iterates through lines

Splits into words

Returns number of characters

Prints answers

To test, you can run this against a sample file containing the first paragraph of this chapter's summary, like listing 8.2.

Listing 8.2 `word_count.tst`

```
Python provides a complete set of control flow elements,
including while and for loops, and conditionals.
Python uses the level of indentation to group blocks
of code with control elements.
```

On running `word_count.py`, you'll get the following output:

```
vern@mac:~/quickpythonbook/code $ python3.1 word_count.py
File has 4 lines, 30 words, 189 characters
```

This code can give you an idea of a Python program. There isn't much code, and most of the work gets done in three lines of code in the `for` loop. Most Pythonistas see this conciseness as one of Python's great strengths.

8.8 **Summary**

Python provides a complete set of control-flow elements, including `while` and `for` loops and conditionals. Python uses the level of indentation to group blocks of code with control elements.

Python has the Boolean values `True` and `False`, which can be referenced by variables, but it also considers any `0` or empty value to be false and any nonzero or non-empty value to be true.

Control flow is an important part of programming, but just as important is the ability to package and reuse blocks of code. In the next few chapters, we'll look at ways to do that in Python, beginning with functions.

THE Quick Python Book SECOND EDITION

Vernon L. 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 **Vern Ceder** is Director of Technology at the Canterbury School in Fort Wayne, Indiana where he teaches and uses Python. 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



9 781935 182207