

VISUAL QUICKSTART GUIDE



Python

Third Edition

TOBY DONALDSON

© LEARN THE QUICK AND EASY WAY!

VISUAL QUICKSTART GUIDE

Python

TOBY DONALDSON

Visual QuickStart Guide

Python, Third Edition

Toby Donaldson

Peachpit Press

www.peachpit.com

To report errors, please send a note to errata@peachpit.com

Peachpit Press is a division of Pearson Education

Copyright © 2014 by Toby Donaldson

Editor: Scout Festa

Production Editor: Katerina Malone

Compositor: David Van Ness

Indexer: Valerie Haynes Perry

Cover Design: RHDG / Riezebos Holzbaur Design Group, Peachpit Press

Interior Design: Peachpit Press

Logo Design: MINE™ www.minesf.com

Notice of Rights

All rights reserved. No part of this book may be reproduced or transmitted in any form by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of the publisher. For information on getting permission for reprints and excerpts, contact permissions@peachpit.com.

Notice of Liability

The information in this book is distributed on an “As Is” basis, without warranty. While every precaution has been taken in the preparation of the book, neither the author nor Peachpit shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the instructions contained in this book or by the computer software and hardware products described in it.

Trademarks

Visual QuickStart Guide is a registered trademark of Peachpit Press, a division of Pearson Education.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and Peachpit was aware of a trademark claim, the designations appear as requested by the owner of the trademark. All other product names and services identified throughout this book are used in editorial fashion only and for the benefit of such companies with no intention of infringement of the trademark. No such use, or the use of any trade name, is intended to convey endorsement or other affiliation with this book.

ISBN-13: 978-0-321-92955-6

ISBN-10: 0-321-92955-1

9 8 7 6 5 4 3 2 1

Printed and bound in the United States of America

Acknowledgments

Thanks to Clifford Colby and Scout Festa for their expertise and patience in bringing this edition of the book to life; to the many students at SFU who continue to teach me how best to learn Python; to John Edgar and the other computer science teachers at SFU with whom I've had the pleasure to work; and to Bonnie, Thomas, and Emily for recommending I avoid using the word *blithering* more than once in these acknowledgments. And a special thank you to Guido van Rossum and the rest of the Python community for creating a programming language that is so much fun to use.

Contents at a Glance

Chapter 1	Introduction to Programming	1
Chapter 2	Arithmetic, Strings, and Variables	9
Chapter 3	Writing Programs	31
Chapter 4	Flow of Control	43
Chapter 5	Functions	67
Chapter 6	Strings	83
Chapter 7	Data Structures	101
Chapter 8	Input and Output	123
Chapter 9	Exception Handling	143
Chapter 10	Object-Oriented Programming	153
Chapter 11	Case Study: Text Statistics	177
Appendix A	Popular Python Packages	195
Appendix B	Comparing Python 2 and Python 3	199
	Index	203

Table of Contents

Chapter 1	Introduction to Programming	1
	The Python Language	2
	What Is Python Useful For?	3
	How Programmers Work	4
	Installing Python	6
Chapter 2	Arithmetic, Strings, and Variables	9
	The Interactive Command Shell	10
	Integer Arithmetic	11
	Floating Point Arithmetic	13
	Other Math Functions	16
	Strings	17
	String Concatenation	19
	Getting Help	20
	Converting Between Types	22
	Variables and Values	24
	Assignment Statements	26
	How Variables Refer to Values	28
	Multiple Assignment	29
Chapter 3	Writing Programs	31
	Using IDLE's Editor	32
	Compiling Source Code	35
	Reading Strings from the Keyboard	36
	Printing Strings on the Screen	39
	Source Code Comments	41
	Structuring a Program	42
Chapter 4	Flow of Control	43
	Boolean Logic	44
	If-Statements	49
	Code Blocks and Indentation	51
	Loops	54

	Comparing For-Loops and While-Loops	59
	Breaking Out of Loops and Blocks	64
	Loops Within Loops	66
Chapter 5	Functions	67
	Calling Functions	68
	Defining Functions	70
	Variable Scope	73
	Using a main Function	75
	Function Parameters	76
	Modules	80
Chapter 6	Strings	83
	String Indexing	84
	Characters	87
	Slicing Strings	89
	Standard String Functions	92
	Regular Expressions	98
Chapter 7	Data Structures	101
	The type Command	102
	Sequences	103
	Tuples	104
	Lists	108
	List Functions	110
	Sorting Lists	113
	List Comprehensions	115
	Dictionaries	118
	Sets	122
Chapter 8	Input and Output	123
	Formatting Strings	124
	String Formatting	126
	Reading and Writing Files	128
	Examining Files and Folders	131
	Processing Text Files	134
	Processing Binary Files	138
	Reading Webpages	141

Chapter 9	Exception Handling	143
	Exceptions	144
	Catching Exceptions	146
	Clean-Up Actions	150
Chapter 10	Object-Oriented Programming	153
	Writing a Class	154
	Displaying Objects	156
	Flexible Initialization	160
	Setters and Getters	162
	Inheritance	168
	Polymorphism	171
	Learning More	175
Chapter 11	Case Study: Text Statistics	177
	Problem Description	178
	Keeping the Letters We Want	180
	Testing the Code on a Large Data File	182
	Finding the Most Frequent Words	184
	Converting a String to a Frequency Dictionary	187
	Putting It All Together	188
	Exercises	190
	The Final Program	192
Appendix A	Popular Python Packages	195
	Some Popular Packages	196
Appendix B	Comparing Python 2 and Python 3	199
	What's New in Python 3	200
	Index	203

This page intentionally left blank

4

Flow of Control

The programs we've written so far are *straight-line* programs that consist of a sequence of Python statements executed one after the other. The flow of execution is simply a straight sequence of statements, with no branching or looping back to previous statements.

In this chapter, we look at how to change the order in which statements are executed by using if-statements and loops. Both are essential in almost any nontrivial program.

Both if-statements and loops are controlled by logical expressions, and so the first part of this chapter will introduce the idea of Boolean logic.

Read the sample programs in this chapter carefully. Take the time to try them out and make your own modifications.

In This Chapter

Boolean Logic	44
If-Statements	49
Code Blocks and Indentation	51
Loops	54
Comparing For-Loops and While-Loops	59
Breaking Out of Loops and Blocks	64
Loops Within Loops	66

Boolean Logic

In Python, as in most programming languages, decisions are made using *Boolean logic*. Boolean logic is all about manipulating so-called truth values, which in Python are written **True** and **False**. Boolean logic is simpler than numeric arithmetic, and is a formalization of logical rules you already know.

We combine Boolean values using four main *logical operators* (or *logical connectives*): **not**, **and**, **or**, and **==**. All decisions that can be made by Python—or any computer language, for that matter—can be made using these logical operators.

Suppose that **p** and **q** are two Python variables each labeling Boolean values. Since each has two possible values (**True** or **False**), altogether there are four different sets of values for **p** and **q** (see the first two columns of **Table 4.1**). We can now define the logical operators by specifying exactly what value they return for the different truth values of **p** and **q**. These kinds of definitions are known as *truth tables*, and Python uses an internal version of them to evaluate Boolean expressions.

TABLE 4.1 Truth Table for Basic Logical Operators

p	q	p == q	p != q	p and q	p or q	not p
False	False	True	False	False	False	True
False	True	False	True	False	True	True
True	False	False	True	False	True	False
True	True	True	False	True	True	False

Logical equivalence

Let's start with `==`. The expression `p == q` is **True** only when `p` and `q` both have the same truth value—that is, when `p` and `q` are either both **True** or both **False**. The expression `p != q` tests if `p` and `q` are not the same, and returns **True** only when they have different values.

```
>>> False == False
True
>>> True == False
False
>>> True == True
True
>>> False != False
False
>>> True != False
True
>>> True != True
False
```

Logical "and"

The Boolean expression `p and q` is **True** only when both `p` is **True** and `q` is **True**. In every other case it is **False**. The fifth column of Table 4.1 summarizes each case.

```
>>> False and False
False
>>> False and True
False
>>> True and False
False
>>> True and True
True
```

Logical "or"

The Boolean expression `p or q` is **True** exactly when `p` is **True** or `q` is **True**, or when both are **True**. This is summarized in the sixth column of Table 4.1. The only slightly tricky case is when both `p` and `q` are **True**. In this case, the expression `p or q` is **True**.

```
>>> False or False
False
>>> False or True
True
>>> True or False
True
>>> True or True
True
```

Logical negation

Finally, the Boolean expression `not p` is **True** when `p` is **False**, and **False** when `p` is **True**. It essentially *flips* the value of the variable.

```
>>> not True
False
>>> not False
True
```

Evaluating larger Boolean expressions

Since Boolean expressions are used to control both if-statements and loops, it is important to understand how they are evaluated. Just as with arithmetic expressions, Boolean expressions use both brackets and operator precedence to specify the order in which their sub-parts are evaluated.

To evaluate a Boolean expression with brackets:

Suppose we want to evaluate the expression **not (True and (False or True))**. We can do it by following these steps:

- **not (True and (False or True))**

Expressions in brackets are always evaluated first, and so we first evaluate **False or True**, which is **True**.

This makes the original expression equivalent to this simpler one:

not (True and True).

- **not (True and True)**

To evaluate this expression, we again evaluate the expression in brackets first: **True and True** evaluates to **True**, which gives us the equivalent expression: **not True**.

- **not True**

Finally, to evaluate this expression, we simply look up the answer in the last column of Table 4.1: **not True** evaluates to **False**. Thus, the entire expression **not (True and (False or True))** evaluates to **False**. You can easily check that this is the correct answer in Python itself:

```
>>> not (True and (False or
→ True))
False
```

TABLE 4.2 Boolean Operator Priority
(Highest to Lowest)

`p == q`

`p != q`

`not p`

`p and q`

`p or q`

To evaluate a Boolean expression without brackets:

Suppose we want to evaluate the expression **not True and False or True**. This is the same as the previous one, but this time there are no brackets.

■ not True and False or True

We first evaluate the operator with the highest precedence, as listed in **Table 4.2**. In this case, **not** has the highest precedence, and so **not True** is evaluated first (the fact that it happens to be at the start of the expression is a coincidence). This simplifies the expression to **False and False or True**.

■ False and False or True

We again evaluate the operator with the highest precedence. According to **Table 4.2**, **and** has higher precedence than **or**, and so **False and True** is evaluated first. The expression simplifies to **False or True**.

■ False or True

This final expression evaluates to **True**, which is found by looking up the answer in **Table 4.1**. Thus the original expression, **False and not False or True**, evaluates to **True**.

TIP Writing complicated Boolean expressions without brackets is usually a bad idea because they are hard to read and evaluate—not all programmers remember the order of precedence of Boolean operators!

TIP One exception is when you use the *same* logical operator many times in a row. Then it is usually easier to read without the brackets. For example:

```
>>> (True or (False or (True or
→ False)))
```

```
True
```

```
>>> True or False or True or False
True
```

Short-circuit evaluation

The definition of the logical operators given in Table 4.1 is the standard definition you would find in any logic textbook. However, like most modern programming languages, Python uses a simple trick called *short-circuit evaluation* to speed up the evaluation of some Boolean expressions.

Consider the Boolean expression **False and X**, where **X** is *any* Boolean expression. It turns out that no matter whether **X** is **True** or **X** is **False**, the entire expression is **False**. The reason is that the initial **False** makes the whole **and**-expression **False**. The value of **False and X** does not depend on **X**—it is *always False*. In such cases, Python does not evaluate **X** at all—it simply stops and returns the value **False**. This can speed up the evaluation of Boolean expressions.

Similarly, Boolean expressions of the form **True or X** are *always True*, no matter the value of **X**. The precise rules for how Python does short-circuiting are given in Table 4.3.

Most of the time you can ignore short-circuiting and just reap its performance benefits. However, it is useful to remember that Python does this, since every once in a while it could be the source of a subtle bug.

TIP It's possible to use the definitions of **and** and **or** from Table 4.3 to write short and tricky code that simulates **if**-statements (which we will see in the next section). However, such expressions are usually quite difficult to read, so if you ever run across such expressions in other people's Python code (*you should never put anything so ugly in your programs!*), you may need to refer to Table 4.3 to figure out exactly what they are doing.

TABLE 4.3 Definition of Boolean Operators in Python

Operation	Result
p or q	if p is False , then q , else p
p and q	if p is False , then p , else q

If-Statements

If-statements let you change the flow of control in a Python program. Essentially, they let you write programs that can decide, while the programming is running, whether or not to run one block of code or another. Almost all nontrivial programs use one or more if-statements, so they are important to understand.

If/else-statements

Suppose you are writing a password-checking program. The user enters their password, and if it is correct, you log them in to their account. If it is not correct, then you tell them they've entered the wrong password:

```
# password1.py
pwd = input('What is the password? ')
if pwd == 'apple': # note use of == #
    → instead of =
    print('Logging on ...')
else:
    print('Incorrect password.')
print('All done!')
```

It's pretty easy to read this program: If the string that `pwd` labels is `'apple'`, then a login message is printed. But if `pwd` is anything other than `'apple'`, the message *incorrect password* is printed.

An if-statement always begins with the keyword `if`. It is then (always) followed by a Boolean expression called the *if-condition*, or just *condition* for short. After the if-condition comes a colon (:). As we will see, Python uses the `:` token to mark the end of conditions in if-statements, loops, and functions.

Everything from the `if` to the `:` is referred to as the *if-statement* header. If the condition in the header evaluates to `True`, then the statement `print('Logging on ...')` is immediately executed, and `print('Incorrect password.')` is skipped and never executed.

If the condition in the header evaluates to `False`, then `print('Logging on ...')` is skipped, and only the statement `print('Incorrect password.')` is executed.

In all cases, the final `print('All done!')` statement is executed.

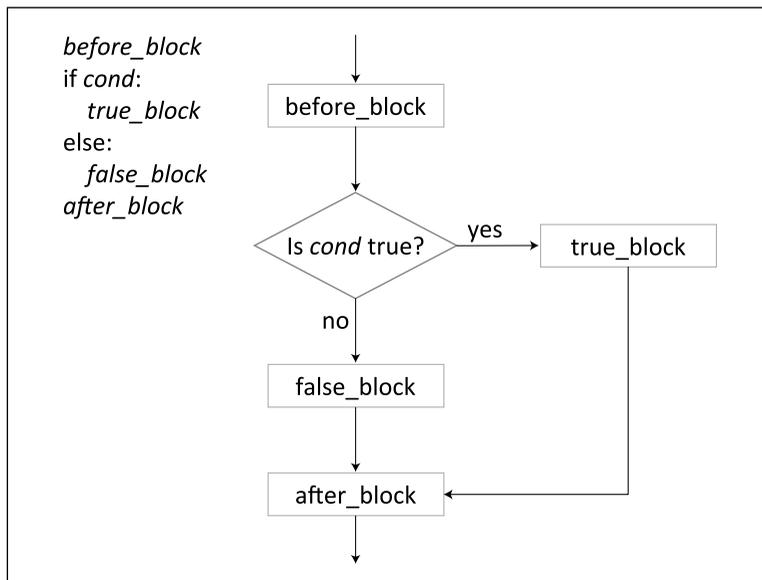
The general structure of an if/else-statement is shown in **A**.

TIP We will often refer to the entire multiline if structure as a single if-statement.

TIP You must put at least one space after the `if` keyword.

TIP The `if` keyword, the condition, and the terminating `:` must appear all on one line without breaks.

TIP The else-block of an if-statement is optional. Depending on the problem you are solving, you may or may not need one.



A This flow chart shows the general format and behavior of an if/else-statement. The code blocks can consist of any number of Python statements (even other if-statements!).

Code Blocks and Indentation

One of the most distinctive features of Python is its use of indentation to mark blocks of code. Consider the if-statement from our password-checking program:

```
if pwd == 'apple':
    print('Logging on ...')
else:
    print('Incorrect password.')
print('All done!')
```

The lines `print('Logging on ...')` and `print('Incorrect password.')` are two separate *code blocks*. These ones happen to be only a single line long, but Python lets you write code blocks consisting of any number of statements.

To indicate a block of code in Python, you must indent each line of the block by the same amount. The two blocks of code in our example if-statement are both indented four spaces, which is a typical amount of indentation for Python.

In most other programming languages, indentation is used only to help make the code look pretty. But in Python, it is required for indicating what block of code a statement belongs to. For instance, the final `print('All done!')` is *not* indented, and so is *not* part of the else-block.

Programmers familiar with other languages often bristle at the thought that indentation matters: Many programmers like the freedom to format their code how they please. However, Python's indentation rules follow a style that many programmers already use to make their code readable. Python simply takes this idea one step further and gives meaning to the indentation.

TIP IDLE is designed to automatically indent code for you. For instance, pressing Return after typing the `:` in an if-header automatically indents the cursor on the next line.

TIP The amount of indentation matters: A missing or extra space in a Python block could cause an error or unexpected behavior. Statements within the same block of code need to be indented at the same level.

If/elif-statements

An if/elif-statement is a generalized if-statement with more than one condition. It is used for making complex decisions. For example, suppose an airline has the following “child” ticket rates: Kids 2 years old or younger fly for free, kids older than 2 but younger than 13 pay a discounted child fare, and anyone 13 years or older pays a regular adult fare. This program determines how much a passenger should pay:

```
# airfare.py
age = int(input('How old are you? '))
if age <= 2:
    print(' free')
elif 2 < age < 13:
    print(' child fare')
else:
    print('adult fare')
```

After Python gets **age** from the user, it enters the if/elif-statement and checks each condition one after the other in the order they are given. So first it checks if **age** is less than 2, and if so, it indicates that the flying is free and jumps out of the elif-condition. If **age** is not less than 2, then it checks the next elif-condition to see if **age** is between 2 and 13. If so, it prints the appropriate message and jumps out of the if/elif-statement. If neither the if-condition nor the elif-condition is **True**, then it executes the code in the else-block.

TIP **elif** is short for *else if*, and you can use as many elif-blocks as needed.

TIP Each of the code blocks in an if/elif-statement must be consistently indented the same amount.

TIP As with a regular if-statement, the else-block is optional. In an if/elif-statement *with* an else-block, *exactly one* of the if/elif-blocks will be executed. If there is no else-block, then it is possible that none of the conditions are **True**, in which case none of the if/elif-blocks are executed.

Conditional expressions

Python has one more logical operator that some programmers like (and some don't!). It's essentially a shorthand notation for if-statements that can be used directly within expressions. Consider this code:

```
food = input("What's your favorite  
→ food? ")  
reply = 'yuck' if food == 'lamb'  
→ else 'yum'
```

The expression on the right-hand side of = in the second line is called a *conditional expression*, and it evaluates to either 'yuck' or 'yum'. It's equivalent to the following:

```
food = input("What's your favorite  
→ food? ")  
if food == 'lamb':  
    reply = 'yuck'  
else:  
    reply = 'yum'
```

Conditional expressions are usually shorter than the corresponding if/else-statements, although not always as flexible or easy to read. In general, you should use them when they make your code simpler.

Loops

Now we turn to loops, which are used to repeatedly execute blocks of code. Python has two main kinds of loops: *for-loops* and *while-loops*. For-loops are generally easier to use and less error prone than while-loops, although not quite as flexible.

For-loops

The basic for-loop repeats a given block of code some specified number of times. For example, this snippet of code prints the numbers 0 to 9 on the screen:

```
# count10.py
for i in range(10):
    print(i)
```

The first line of a for-loop is called the *for-loop header*. A for-loop always begins with the keyword **for**. After that comes the *loop variable*, in this case **i**. Next is the keyword **in**, typically (but not always) followed by **range(n)** and a terminating **:** token. A for-loop repeats its *body*, the code block underneath it, exactly **n** times.

Each time the loop executes, the loop variable **i** is set to be the next value. By default, the initial value of **i** is 0, and it goes up to **n - 1** (not **n!**) by ones. Starting numbering at 0 instead of 1 might seem unusual, but it is common in programming.

If you want to change the starting value of the loop, add a starting value to **range**:

```
for i in range(5, 10):
    print(i)
```

This prints the numbers from 5 to 9.

Lingo Alert

Programmers often use the variable **i** because it is short for *index*, and is also commonly used in mathematics. When we start using loops within loops, it is common to use **j** and **k** as other loop variable names.

TIP If you want to print the numbers from 1 to 10 (instead of 0 to 9), there are two common ways of doing so. One is to change the start and end of the range:

```
for i in range(1, 11):  
    print(i)
```

Or, you can add 1 to *i* inside the loop body:

```
for i in range(10):  
    print(i + 1)
```

TIP If you would like to print numbers in *reverse order*, there are again two standard ways of doing so. The first is to set the range parameters like this:

```
for i in range(10, 0, -1):  
    print(i)
```

Notice that the first value of `range` is 10, the second value is 0, and the third value, called the *step*, is -1. Alternatively, you can use a simpler range and modify *i* in the loop body:

```
for i in range(10):  
    print(10 - i)
```

TIP For-loops are actually more general than described in this section: They can be used with any kind of *iterator*, which is a special kind of programming object that returns values. For instance, we will see later that for-loops are the easiest way to read the lines of a text file.

While-loops

The second kind of Python loop is a *while-loop*. Consider this program:

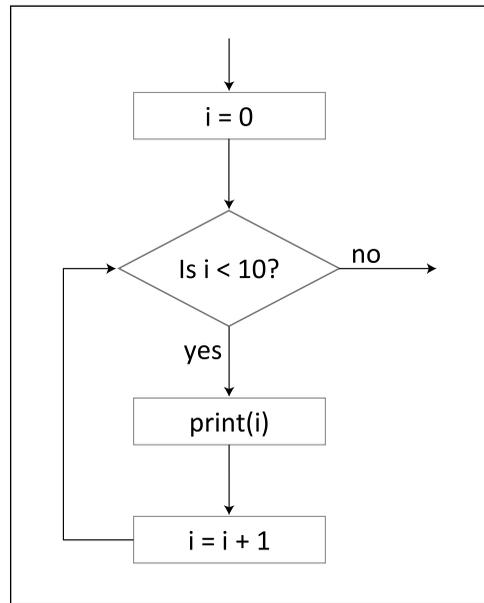
```
# while10.py
i = 0
while i < 10:
    print(i)
    i = i + 1 # add 1 to i
```

This prints out the numbers from 0 to 9 on the screen. It is noticeably more complicated than a for-loop, but it is also more flexible.

The while-loop itself begins on the line beginning with the keyword **while**; this line is called the *while-loop header*, and the indented code underneath it is called the *while-loop body*. The header always starts with **while** and is followed by the *while-loop condition*. The condition is a Boolean expression that returns **True** or **False**.

The flow of control through a while-loop goes like this: First, Python checks if the loop condition is **True** or **False**. If it's **True**, it executes the body; if it's **False**, it skips over the body (that is, it jumps out of the loop) and runs whatever statements appear afterward. When the condition is **True**, the body is executed, and then Python checks the condition again. As long as the loop condition is **True**, Python keeps executing the loop. **B** shows a flow chart for this program.

The very first line of the sample program is `i = 0`, and in the context of a loop it is known as an *initialization statement*, or an *initializer*. Unlike with for-loops, which automatically initialize their loop variable, it is the programmer's responsibility to give initial values to any variables used by a while-loop.

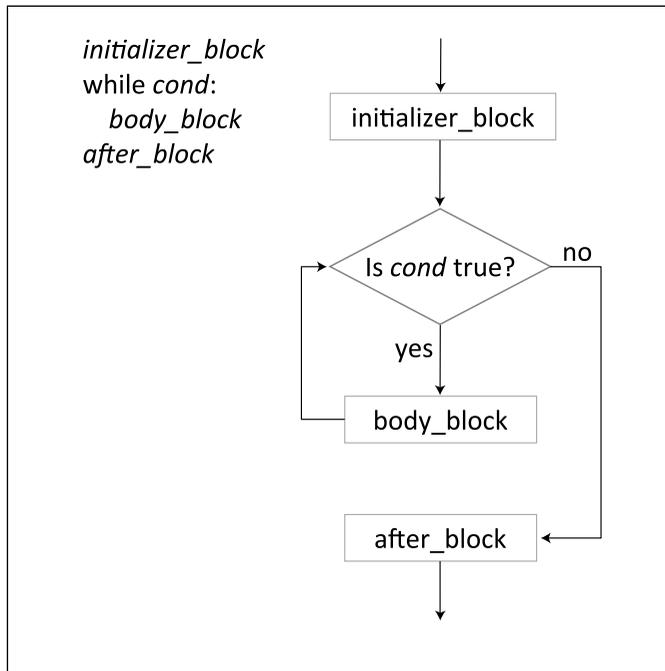


B This is a flow chart for code that counts from 0 to 9. Notice that when the loop condition is **False** (that is, the *no* branch is taken in the decision box), the arrow does not go into a box. That's because in our sample code there is nothing after the while-loop.

The last line of the loop body is $i = i + 1$. As it says in the source code comment, this line causes i to be incremented by 1. Thus, i increases as the loop executes, which guarantees that the loop will eventually stop. In the context of a while-loop, this line is called an *increment*, or *incrementer*, since its job is to increment the loop variable.

The general form of a while-loop is shown in the flow chart of **C**.

Even though almost all while-loops need an initializer and an incrementer, Python does not require that you include them. It is entirely up to you, the programmer, to remember these lines. Even experienced programmers find that while-loop initializers and incrementers are a common source of errors.



C A flow chart for the general form of a while-loop. Note that the incrementer is not shown explicitly: It is embedded somewhere in **body_block**, often (but not always) at the end of that block.

TIP While-loops are *extremely* flexible. You can put any code whatsoever before a while-loop to do whatever kind of initialization is necessary. The loop condition can be *any* Boolean expression, and the incremter can be put *anywhere* within the while-loop body, and it can do whatever you like.

TIP A loop that never ends is called an *infinite loop*. For instance, this runs forever:

```
while True:  
    print('spam')
```

TIP Some programmers like to use infinite loops as a quick way to write a loop. However, this is generally considered to be poor style because such loops often become complex and hard to understand.

TIP Many Python programmers try to use for-loops whenever possible and use while-loops only when absolutely necessary.

TIP While-loops can be written with an else-block. However, this unusual feature is rarely used in practice, so we haven't discussed it. If you are curious, you can read about it in the online Python documentation—for example, http://docs.python.org/3/reference/compound_stmts.html.

Comparing For-Loops and While-Loops

Let's take a look at a few examples of how for-loops and while-loops can be used to solve the same problems. Plus we'll see a simple program that can't be written using a for-loop.

Calculating factorials

Factorials are numbers of the form $1 \times 2 \times 3 \times \dots \times n$, and they tell you how many ways n objects can be arranged in a line. For example, the letters ABCD can be arranged in $1 \times 2 \times 3 \times 4 = 24$ different ways. Here's one way to calculate factorials using a for-loop:

```
# forfact.py
n = int(input('Enter an integer
-> >= 0: '))
fact = 1
for i in range(2, n + 1):
    fact = fact * i
print(str(n) + ' factorial is ' +
-> str(fact))
```

Here's another way to do it using a while-loop:

```
# whilefact.py
n = int(input('Enter an integer
-> >= 0: '))
fact = 1
i = 2
while i <= n:
    fact = fact * i
    i = i + 1
print(str(n) + ' factorial is ' +
-> str(fact))
```

continues on next page

Both of these programs behave the same from the user's perspective, but the internals are quite different. As is usually the case, the while-loop version is a little more complicated than the for-loop version.

TIP In mathematics, the notation $n!$ is used to indicate factorials. For example, $4! = 1 \times 2 \times 3 \times 4 = 24$. By definition, $0! = 1$. Interestingly, there is *no* simple formula for calculating factorials.

TIP Python has no maximum integer, so you can use these programs to calculate very large factorials. For example, a deck of cards can be arranged in exactly $52!$ ways:

```
Enter an integer >= 0: 52
52 factorial is 80658175170943878571
→ 6606368564037669752895054408832778
→ 2400000000000
```

Summing numbers from the user

The following programs ask the user to enter some numbers, and then prints their sum. Here is a version using a for-loop:

```
# forsum.py
n = int(input('How many numbers to
→ sum? '))
total = 0
for i in range(n):
    s = input('Enter number ' +
→ str(i + 1) + ': ')
    total = total + int(s)
print('The sum is ' + str(total))
```

Here's a program that does that same thing using a while-loop:

```
# whilesum.py
n = int(input('How many numbers to
→ sum? '))
total = 0
i = 1
while i <= n:
    s = input('Enter number ' +
→ str(i) + ': ')
    total = total + int(s)
    i = i + 1
print('The sum is ' + str(total))
```

Again, the while-loop version is a little more complex than the for-loop version.

TIP These programs assume that the user is entering integers. Floating point numbers will be truncated when `int(s)` is called. Of course, you can easily change this to `float(s)` if you want to allow floating point numbers.

Summing an unknown number of numbers

Now here's something that can't be done with the for-loops we've seen so far. Suppose we want to let users enter a list of numbers to be summed without asking them ahead of time how many numbers they have. Instead, they just type **'done'** when they have no more numbers to add. Here's how to do it using a while-loop:

```
# donesum.py
total = 0
s = input('Enter a number (or
→ "done"): ')
while s != 'done':
    num = int(s)
    total = total + num
    s = input('Enter a number (or
→ "done"): ')
print('The sum is ' + str(total))
```

The idea here is to keep asking users to enter a number, quitting only when they enter **'done'**. The program doesn't know ahead of time how many times the loop body will be executed.

Notice a few more details:

- We must call **input** in two different places: before the loop and inside the loop body. This is necessary because the loop condition decides whether or not the input is a number or **'done'**.
- The ordering of the statements in the loop body is very important. If the loop condition is **True**, then we know **s** is not **'done'**, and so we assume it is an integer. Thus we can convert it to an integer, add it to the running total, and then ask the user for more input.
- We convert the input string **s** to an integer only *after* we know **s** is not the string **'done'**. If we had written

```
s = int(input('Enter a number  
→ (or "done"): '))
```

as we had previously, the program would crash when the user typed **'done'**.

- There is no need for the **i** counter variable anymore. In the previous summing programs, **i** tracked how many numbers had been entered so far. As a general rule, a program with fewer variables is easier to read, debug, and extend.

Breaking Out of Loops and Blocks

The **break** statement is a handy way for exiting a loop from anywhere within the loop's body. For example, here is an alternative way to sum an unknown number of numbers:

```
# donesum_break.py
total = 0
while True:
    s = input('Enter a number (or
    → "done"): ')
    if s == 'done':
        break # jump out of the loop
    num = int(s)
    total = total + num
print('The sum is ' + str(total))
```

The while-loop condition is simply **True**, which means it will loop forever unless **break** is executed. The only way for **break** to be executed is if **s** equals **'done'**.

An advantage of this program over **donesum.py** is that the **input** statement is not repeated. But a disadvantage is that the reason for why the loop ends is buried in the loop body. It's not so hard to see it in this small example, but in larger programs **break** statements can be tricky to see. Furthermore, you can have as many **breaks** as you want, which adds to the complexity of understanding the loop.

Generally, it is wise to avoid the **break** statement, and to use it only when it makes your code simpler or clearer.

A relative of **break** is the **continue** statement: When **continue** is called inside a loop body, it immediately jumps up to the loop condition—thus continuing with the next iteration of the loop. It is a little less common than **break**, and generally it should be avoided altogether.

TIP Both **break** and **continue** also work with **for-loops**.

Loops Within Loops

Loops within loops, also known as *nested loops*, occur frequently in programming. For instance, here's a program that prints the times tables up to 10:

```
# timestable.py
for row in range(1, 10):
    for col in range(1, 10):
        prod = row * col
        if prod < 10:
            print(' ', end = '')
            print(row * col, ' ', end = '')
        print()
```

Look carefully at the indentation of the code in this program: It's how you tell what statements belong to what blocks. The final `print()` statement lines up with the second `for`, meaning it is part of the outer for-loop (but not the inner).

Note that the statement `if prod < 10` is used to make the output look neatly formatted. Without it, the numbers won't line up nicely.

TIP When using nested loops, be careful with loop index variables: Do not accidentally reuse the same variable for a different loop. Most of the time, every individual loop needs its own control variables.

TIP You can nest as many loops within loops as you need, although the complexity increases greatly as you do so.

TIP As mentioned previously, if you use `break` or `continue` with nested loops, `break` only breaks out of the innermost loop, and `continue` only “continues” the innermost loop.

This page intentionally left blank

Index

Numbers

2to3 tool, using for Python conversions, 201
5 vs. 5.0, 13

Symbols

'+ file module, meaning of, 134, 137
== operator, 44
+ (addition) operator, 12
= (assignment) operator, example of, 24
\
 (backward slash)
 using with pathnames, 130
 writing, 130
) (closed round bracket), using with
 tuples, 29
% conversion specifier, meaning of, 125
@ (decorators), using, 163
/ (division) operator, 12
// (division) operator, 11
" (double quote), using with strings, 17
__ (double underscore), use of, 20–21
' ' and " " (empty strings), using, 18
\

[] (square brackets)
 using with lists, 108
 using with strings, 84
- (subtraction) operator, 12

A

'a file module, meaning of, 134
addition (+) operator, 12
aggregate data structures, strings as, 83
and operator, 44–45
append function, using with lists, 110–111
append mode, using with text files, 134, 136
area function
 calling, 70–71
 parts of, 71
 return statement, 72
arithmetic operators. *See also* floating point
 arithmetic; integer arithmetic; math
 functions
 addition (+), 12
 division (/), 12
 exponentiation (**), 12
 integer division (//), 11–12
 multiplication (*), 12
 remainder (%), 12
 subtraction (-), 12
ASCII (American Standard Code for Information
 Interchange), 87
assignment (=) operator, example of, 24
assignment statements
 diagrams, 28
 example of, 24
 initialization statement, 26
 labeling values, 28
 left-hand side, 26
 multiple, 29
 operator, 26
 right-hand side, 26
associative arrays. *See* dictionaries

B

- 'b' file module, meaning of, 134, 138
- backward slash (\)
 - using with pathnames, 130
 - writing, 130
- base class, explained, 169–170
- bill.txt** file, using, 182–183, 189
- bin** built-in function, printing doc string for, 21
- binary files, processing, 138–140
- binary mode, indicating, 134
- binary vs. text files, 128–129
- blocks of code
 - breaking out of, 64–65
 - indenting, 51–53
 - indicating, 51
- Boolean logic
 - ==** operator, 44
 - and** operator, 44–45, 48
 - with brackets (**()**), 46
 - definition of operators, 48
 - evaluating expressions, 46–47
 - explained, 44
 - False** values, 44
 - logical equivalence, 45
 - logical negation, 44
 - logical operators, 44
 - not** operator, 44–46
 - operator priority, 47
 - or** operator, 44–45
 - or** operator, 48
 - short-circuit evaluation, 48
 - True** values, 44
 - truth table, 44
 - truth values, 44
 - without brackets (**()**), 47
- Bottle framework, 196
- brackets, preceding word counts with, 186
- break** statement, using, 64–65

C

- calculating
 - area of circle, 70
 - factorials, 59–60
 - powers, 68
- case sensitivity, explained, 25
- case study. See text statistics case study

- case-changing functions
 - s.capitalize()**, 94
 - s.lower()**, 94
 - s.swapcase()**, 94
 - s.title()**, 94
 - s.upper()**, 94
- catching exceptions, 146–149
- ceil(x)** function, 16
- character codes, finding, 87
- character length, determining, 178
- characters
 - accessing with for-loop, 86
 - escape, 88
 - getting rid of unwanted, 180–181
 - whitespace, 88
- Cheetah templating package, 126
- child class, explained, 170
- chr** function, using, 87
- circle, calculating area of, 70
- class diagram, example of, 170
- class hierarchy, example of, 170
- classes
 - defined, 153
 - deriving, 169–170
 - extending, 169–170
 - and methods, 154
 - and objects, 153
 - reusing, 168–170
 - self parameter, 155
 - subclasses of, 169–170
 - writing, 154–155
- clean-up actions
 - finally** code block, 150
 - with** statement, 151
- closed round bracket (**()**), using with tuples, 29
- code blocks
 - breaking out of, 64–65
 - indenting, 51–53
 - indicating, 51
- command line
 - calling Python from, 33–34
 - environment variables, 34
 - path variable, 34
 - running programs from, 33
- command shell
 - interacting with, 10
 - shell prompt, 10

- command window, opening, 34
- comments
 - defined, 36
 - using, 41–42
- compiled code. See object code
- compiling source code, 35
- complex numbers, 15
- concatenating
 - strings, 19
 - tuples, 107
- conditional expressions, 53
- constructors, explained, 154
- continue** statement, using, 64–65
- conversion specifiers
 - % character, 125
 - base 8 value, 125
 - base 16, 125
 - float, 125
 - integers, 125
 - lowercase float exponential, 125
 - lowercase hexadecimal, 125
 - octal value, 125
 - string, 125
 - uppercase hexadecimal, 125
 - uppercase float exponential, 125
- converting
 - floats to integers, 23
 - floats to strings, 22
 - integers to floats, 22
 - integers to strings, 22
 - strings to floats, 22
 - strings to numbers, 23
- cost(x)** function, 16
- count** function, using with lists, 110
- current working directory
 - cwd_size_in_bytes** function, 132
 - explained, 130

D

- d** conversion specifier, meaning of, 125
- data structures
 - defined, 101
 - dictionaries, 118–121
 - list comprehensions, 115–117
 - list functions, 110–112
 - lists, 108–109
 - reading, 139
- self-referential, 109
- sequences, 103
- sets, 122
- sorting lists, 113–114
- tuples, 104–107
- type** command, 102
- writing, 139
- data types
 - checking with **type** command, 102
 - converting between, 22–23
 - converting numeric types, 22
 - explained, 9
 - floats to strings, 22
 - implicit conversions, 22–23
 - integers to floats, 22
 - integers to strings, 22
 - strings, 9
 - strings to floats, 22
- decorators (@), using 163
- degrees(x)** function, 16
- derived class, explained, 169–170
- dictionaries
 - converting to, 184, 187
 - converting to list of tuples, 185–186
 - defined, 118
 - extracting information from, 185
 - key restrictions, 119
 - and sets, 122
 - unique keys, 119
- dictionary functions
 - d.clear()**, 120
 - d.copy()**, 120
 - d.fromkeys()**, 120
 - d.get(key)**, 120
 - d.items()**, 120–121
 - d.keys()**, 120–121
 - d.popitem()**, 120–121
 - d.pop(key)**, 120
 - d.setdefault()**, 120
 - d.update()**, 120
 - d.values()**, 120–121
- dir ('')** command, entering, 37
- dir** function, using, 92
- directory
 - current working, 130–132
 - default, 130
- dir(m)** function, using, 20

display method, using, 157, 159
division (`//` and `/`) operators, 11–12
Django framework, 196
documentation strings
 accessing for functions, 71
 benefits, 71
 formatting convention, 71
 printing, 21
documentation website, accessing, 133
dot notation, using with objects, 155
double quote (`"`), using with strings, 17
double underscore (`_`), use of, 20–21

E

e conversion specifier, meaning of, 125
E conversion specifier, meaning of, 125
Easter egg example, 82
eat_vowels example, 117
editor window, opening in IDLE, 32
elif (else if) statements, 52
else statements, 49–50
empty lists, denoting, 108
empty strings (`' '` and `""`), using, 18, 39
ending lines of text, 88
environment variables, 34
errors, handling, 143
escape characters
 `\`, 88
 `\'`, 88
 `\\`, 88
 `\n`, 88
 `\r`, 88
 `\t`, 88
exceptions
 built-in, 145
 catching, 146–149
 checking for, 146
 defined, 143
 IOError, 143
 outputting tracebacks, 144
 raising, 143–145
 syntax errors, 145
 throwing, 144
executable code. *See* object code
exponentiation (`**`) operator, 12
exp(x) function, 16

F

F conversion specifier, meaning of, 125
factorials, calculating, 59–60
factorial(x) function, 16
file modules, 134
file_stats, calling, 183
files
 examining, 131–133
 functions, 131
 reading, 128–130
 text vs. binary, 128–129
 writing, 128–130
finally code block, adding, 150
find function vs. **index**, 93
float, conversion specifier for, 125
float literals, 13
floating point arithmetic. *See also* arithmetic operators
 5 vs. 5.0, 13
 complex numbers, 15
 decimal points, 13
 errors, 15
 examples, 13
 limited precision, 14–15
 overflow, 14
 scientific notation, 13
 silent errors, 14
 truncating, 61
floats
 converting integers to, 22
 converting strings to, 22
 converting to integers, 23
 converting to strings, 22
float(s) conversions, making, 23
flow of control
 backing out of blocks, 64–65
 backing out of loops, 64–65
 Boolean logic, 44–48
 code blocks, 51–53
 explained, 43
 for-loops vs. while-loops, 59–63
 if-statements, 49–50
 indentation, 51–53
 loops, 54–58
 nested loops, 66

- folders
 - backward slash (\), 130
 - functions, 131
 - pathnames, 130
 - structure, 130
- for-loops
 - accessing characters with, 86
 - changing starting value of, 54
 - headers, 54
 - i** (index) variable, 54, 63
 - printing numbers, 55
 - using iterators with, 55
 - vs. while-loops, 58–63
- format** function
 - using, 94
 - using with strings, 124
- format strings
 - named replacement, 126
 - using, 126–127
 - using curly braces ({}), 127
- formatting functions for strings. *See* string-formatting functions
- formatting parameters, specifying, 127
- f.read()**, calling, 137–138
- frequency dictionaries, converting strings to, 187
- f.seek()**, calling, 137
- function names, reassigning, 69
- function parameters
 - default values, 78
 - keyword parameters, 79
 - pass by reference, 76–77
 - pass by value, 76
 - state of memory, 76
- functional programming style, 72
- functions. *See also* string functions; tuple functions
 - accessing doc strings for, 71
 - append**, 110–111
 - availability to strings, 37
 - as black boxes, 68
 - calculating powers, 68
 - calling, 68–69
 - chr**, 87
 - count**, 110
 - defined, 67
 - defining, 70–72
 - files and folders, 131
 - listing built-in, 21
 - listing in modules, 20
 - main()**, 75
 - vs. methods, 154
 - modules, 80–81
 - naming, 70
 - not returning values, 69
 - ord**, 87
 - side effects, 72
 - using, 133
 - using round brackets (()) with, 68
 - variable scope, 73–74

G

- generator expressions
 - explained, 117
 - searching for, 132
- getters and setters
 - avoiding setters, 167
 - decorators, 163
 - name** and **age** values, 162
 - private variables, 166
 - property decorators, 163–165
 - syntax, 167
 - using, 162–167
- global variables, explained, 74

H

- hapax legomenon, explained, 190
- hash tables. *See* dictionaries
- hashing, using with dictionaries, 118
- help
 - documentation, 21
 - listing functions in modules, 20
 - utility, 21
- help(f)** function, using, 21
- hexadecimal numbers, explained, 138
- Human** class, writing, 169

I

- i** (index) variable, use of, 54, 63
- identifiers, explained, 24
- IDLE (integrated development environment), 4

- IDLE editor
 - alternatives, 33
 - starting screen, 6
 - using, 32–34
- IDLE shortcuts
 - opening editor window, 32
 - opening files for editing, 32
 - redoing last undo, 32
 - running programs, 32
 - saving programs, 32
 - undoing actions in IDLE, 32
- if/elif-statements, 52
- if/else-statements, 49–50
- if-statements
 - explained, 49
 - flow chart, 50
 - headers, 50
 - structure, 50
- immutable objects, 167
- importing
 - modules, 16, 81
 - this** module at command line, 82
- indenting code blocks, 51–53
- index** function vs. **find**, 93
- indexing
 - beginning at 0, 84
 - negative, 85, 91
 - strings, 84–86
 - using **% (mod)** function for, 84
- infinite loops, 58
- inheritance
 - defined, 153, 168–169
 - Human** class, 169
 - isa* terminology, 169
 - overriding methods, 170
 - Player** class, 168–169
- __init__** function, using, 155, 160–161
- initialization, flexibility of, 160–161
- initialization statement, explained, 26
- input** built-in function
 - explained, 36–37
 - using, 123
- installing Python
 - on Linux systems, 7
 - on Macs, 7
 - on Windows systems, 6
- int** function, documentation for, 146
- integer arithmetic. *See also* arithmetic operators; math functions
 - defined, 11
 - division, 11
 - operators, 12
 - order of evaluation, 12
 - unlimited size, 12
- integer division (**//**) operator, 11–12
- integers
 - conversion specifier, 125
 - converting floats to, 23
 - converting to floats, 22
 - converting to strings, 22
 - lack of maximum, 60
- interactive command shell, 10
- interpreter, playing with examples in, 178
- int(s)** conversions, making, 23
- I/O (input and output)
 - console, 123
 - examining files, 131–133
 - examining folders, 131–133
 - explained, 123
 - formatting strings, 124–125
 - processing binary files, 138–140
 - processing text files, 134–137
 - reading files, 128–130
 - reading webpages, 141
 - string formatting, 126–127
 - writing files, 128–130
- IOError**, raising, 143
- isa* terminology, using with inheritance, 169
- iterators, using with for-loops, 55

J

- join** function
 - using, 97
 - using with list comprehensions, 117

K

- keywords
 - restriction for variables, 25
 - using, 79

L

- len** function, using with characters, 178
- letters, keeping desired, 180–181
- lexicographical ordering, 113

- lines of text, ending, 88
- Linux, installing Python on, 7
- list comprehensions
 - examples, 116
 - explained, 115
 - filtering, 117
 - generator expressions, 117
- list functions
 - mutating, 110
 - s.append()**, 110–111
 - s.count()**, 110
 - s.extend()**, 110
 - s.index()**, 110
 - s.insert()**, 110
 - s.pop()**, 110
 - s.remove()**, 110, 112
 - s.reverse()**, 110, 112
 - s.sort()**, 110
- lists. *See also* tuples
 - [] (square brackets), 108
 - containing elements vs. pointing, 109
 - empty, 108
 - lexicographical ordering, 113
 - mutability, 109
 - pointing to values, 109
 - pop and push, 111–112
 - self-referential data structure, 109
 - sorting, 113–114
 - using, 108
- local variable, explained, 73
- log(x)** functions, 16
- loops
 - breaking out of, 64–65
 - infinite loops, 58
 - for-loops, 54–55
 - nesting, 66
 - while-loops, 56–58
- lowercase float exponential, conversion
 - specifier for, 125
- lowercase hexadecimal, conversion specifier
 - for, 125

M

- ^M character, handling, 88
- Macs, installing Python on, 7
- main()** function, using, 75
- maps. *See* dictionaries

- math functions. *See also* arithmetic operators;
 - integer arithmetic
 - importing modules, 16
 - return values, 16
- math** module
 - ceil(x)** function, 16
 - cost(x)** function, 16
 - degrees(x)** function, 16
 - exp(x)** function, 16
 - factorial(x)** function, 16
 - log(x)** functions, 16
 - pow(x)** function, 16
 - radians(x)** function, 16
 - sin(x)** function, 16
 - sqrt(x)** function, 16
 - tan(x)** function, 16
 - using, 16
- methods
 - vs. functions, 154
 - overriding, 170
- mod (%)** function, using with strings, 84
- modules
 - creating, 80
 - importing, 16, 81
 - listing functions in, 20
 - namespaces, 82
 - pickle**, 140
 - shelve**, 140
 - sqlite3**, 140
 - urllib**, 141
 - using, 81
 - webbrowser**, 141
- move** functions, implementing for Undercut
 - game, 172–173
- multiplication (*) operator, 12

N

- \n (newline) character, explained, 39, 88
- n!** notation, using, 60
- name clashes, preventing, 82
- namespaces
 - explained, 82
 - preventing name clashes, 82
- negative indexing, 85, 91
- nested loops
 - break** statement, 66
 - continue** statement, 66
 - using, 66

new keyword, using with constructors, 154
newline (`\n`) character, explained, 39
None value, using with functions, 72
normalize() function, using, 180
not operator, 44–46
number sign (`#`), using with comments, 41
numbers
 converting strings to, 23
 floating point, 38
 immutable quality, 28
 integers, 38
 reading from keyboard, 38
 as strings, 38
 summing, 62
 summing from users, 61
 types of, 38

O

o conversion specifier, meaning of, 125
object code
 converting source code to, 5
 explained, 35
object serialization, explained, 139
objects
 and classes, 153
 creating, 159
 defined, 153
 displaying, 156–159
 dot notation, 155
 immutable, 167
 string representation of, 159
 using, 155
octal values, conversion specifier for, 125
OOP (object-oriented programming), 2
 classes, 153–155
 constructors, 154
 explained, 153
 getters, 162–167
 inheritance, 168–170
 initialization, 160–161
 objects, 156–159
 polymorphism, 171–174
 setters, 162–167
open function
 documentation, 146
 using, 135
open round bracket (`()`), using with tuples, 29

operators. See arithmetic operators;
 assignment operator
or operator, 44–45
ord function, using with character codes, 87
order of evaluation, 12
ordered sequences, 103
os.chdir() function, 131
os.getcwd() function, 131
os.listdir() function, 131
os.path.isdir() function, 131
os.path.isfile() function, 131
os.stat() function, 131, 133
overflow errors, 14

P

packages
 Bottle, 196
 Django, 196
 PIL (Python Imaging Library), 196
 Pygame, 197
 PyPI (Python Package Index), 197
 SciPy, 197
 Tkinter, 196
 Twisted, 197
parent class, explained, 170
partition function, using, 95
pass by reference, explained, 76
pass by value, explained, 76
path variable, 34
pathnames, using with folders, 130
Person class
 adding **method** to, 156
 creating, 154
Person objects
 with **name** and **age**, 160–161
 working with, 158
pi calculation, doing, 70
pickle module
 restriction, 140
 using, 139
PIL (Python Imaging Library) package, 196
play_undercut function, analyzing, 174
Player class, creating, 168
polymorphism
 defined, 153
 power of, 174
 Undercut game, 171–174
pop, using on lists, 111–112

- powers, calculating, 68
- pow(x)** function, 16
- print** statement
 - using, 39–40, 135
 - using string interpolation with, 151
- printing
 - documentation strings, 21
 - numbers in for-loops, 55
 - strings on screen, 39–40
- private variables, 166–167
- problems, understanding, 178
- programming
 - process, 4–5
 - requirements, 4
 - source code, 5
- programming problems, understanding, 178
- programs
 - checking output, 5
 - defined, 31
 - flow of execution, 43
 - managing variables, 167
 - running, 5
 - running from command line, 33
 - running with IDLE, 32
 - storing, 32
 - straight-line, 43
 - structuring, 42
 - tracing, 36–37
 - writing in IDLE, 32
- property decorators, using, 163–165
- public variables, 166
- push**, using on lists, 111
- .py** files
 - versus **.pyc** files, 35
 - contents of, 5
 - listing, 132
 - running, 35
- .pyc** files
 - contents, 35
 - explained, 4
- Pygame 2D animation package, 197
- PyPI (Python Package Index) package, 197
- Python 2
 - classes, 200
 - converting into Python 3, 201
 - dividing integers, 200
 - vs. Python 3, 40, 200–201
 - raw_input** function, 200
 - string interpolation, 200
 - xrange** function, 200
- Python 3
 - dividing integers, 200
 - format strings, 200
 - input** function, 200
 - print** function, 200
 - range** function, 200
- Python components
 - compiler, 35
 - interpreter, 35
 - virtual machine, 35
- Python language
 - calling from command line, 33–34
 - design, 2
 - download page, 6
 - education, 3
 - installing on Linux, 7
 - installing on Macs, 7
 - installing on Windows, 6
 - libraries, 2
 - maintainability, 2
 - origin of name, 2
 - scientific computing, 3
 - scripts, 3
 - text processing, 3
 - uses, 3
 - website development, 3
- Python packages
 - Bottle, 196
 - Django, 196
 - PIL (Python Imaging Library), 196
 - Pygame, 197
 - PyPI (Python Package Index), 197
 - SciPy, 197
 - Tkinter, 196
 - Twisted, 197
- pythonintro website, accessing, 133

Q

- quotes (' and "), using with strings, 17
- quotes, triple, 17

R

- '**r**' file module, meaning of, 134
- \r** escape character, 88
- radians(x)** function, 16

- re** module, accessing documentation for, 100
- reading
 - files, 128–130
 - text files as strings, 135
 - webpages, 141
- regular expressions
 - examples, 98–99
 - matching with, 99
 - operators, 98
 - using, 181
 - using round brackets (()) with, 99
 - x*** operator, 98
 - x|y** operator, 98
 - x+** operator, 98
 - xy?** operator, 98
- remainder (%) operator, 12
- remove** function, using with lists, 112
- replace** function, using with strings, 96, 180
- __repr__** method, using, 158–159
- return** statement, using with **area** function, 72
- return values, using, 16
- reverse** function, using with lists, 112
- round brackets (())
 - using with functions, 68
 - using with regular expressions, 99
 - using with tuples, 104
- rpartition** function, using, 95

S

- s** conversion specifier, meaning of, 125
- saving programs with IDLE, 32. *See also* IDLE (integrated development environment)
- scientific notation, using, 13
- SciPy scientific computing package, 197
- scope. *See* variable scope
- scripts. *See* programs
- searching functions for strings. *See* string-searching functions
- self parameter, using with classes, 155
- sentences, splitting into words, 179
- sequence types
 - lists, 103
 - strings, 103
 - tuples, 103–107
- sequences. *See also* values
 - defined, 103
 - ordered, 103

- size restriction, 103
- serialization, explained, 139
- sessions. *See* shell transcripts
- sets
 - calling **dir(set)**, 122
 - and dictionaries, 122
 - explained, 122
 - immutable frozensets, 122
 - mutable, 122
 - online documentation, 122
- setters and getters
 - avoiding setters, 167
 - decorators, 163
 - name** and **age** values, 162
 - private variables, 166
 - property decorators, 163–165
 - syntax, 167
 - using, 162–167
- shell prompt (>>>), 10
- shell transcript, explained, 10
- shelve** module, explained, 140
- side effects, relationship to functions, 72
- sin(x)** function, 16
- single quote ('), using with strings, 17
- slicing strings
 - explained, 89
 - with negative indexes, 91
 - shortcuts, 90–91
- software. *See* object code
- sort** function, using with lists, 114
- sorting
 - lists, 113–114
 - tuples, 114
- source code
 - comments, 36, 41–42
 - compiling, 35
 - converting to object code, 5
 - writing, 5
- split** function, using, 96, 178–179
- splitting functions for strings. *See* string-splitting functions
- sqlite3** module, explained, 140
- sqrt(x)** function, 16
- square brackets ([])
 - using with lists, 108
 - using with strings, 84
- standard error (stderr), explained, 39

- standard input (stdin), explained, 39
- standard output (stdout), explained, 39
- stop words, creating set of, 190
- string functions. *See also* functions
 - case-changing, 94
 - for contents of substrings, 92
 - s.count()**, 97
 - for searching, 93
 - s.encode()**, 97
 - s.endswith()**, 92
 - s.find()**, 93
 - s.index()**, 93
 - s.isalnum()**, 92
 - s.isalpha()**, 92
 - s.isdecimal()**, 92
 - s.isdigit()**, 92
 - s.isidentifier()**, 92
 - s.islower()**, 92
 - s.isnumeric()**, 92
 - s.isprintable()**, 92
 - s.isspace()**, 92
 - s.istitle()**, 92
 - s.isupper()**, 92
 - s.join()**, 97
 - s.maketrans()**, 97
 - split**, 95–96
 - s.rfind()**, 93
 - s.rindex()**, 93
 - s.startswith()**, 92
 - s.translate()**, 97
 - for stripping, 95–96
 - s.zfill()**, 97
 - for testing, 92
- string interpolation, 124, 151
- string literals, writing, 17
- string-formatting functions
 - s.center()**, 94
 - s.format()**, 94
 - s.ljust()**, 94
 - s.rjust()**, 94
- string-replacement functions
 - s.expandtabs()**, 96
 - s.replace()**, 96
- strings
 - as aggregate data structures, 83
 - characters, 86–88
 - concatenating, 19
 - conversion specifiers, 125
 - converting floats to, 22
 - converting integers to, 22
 - converting to floats, 22
 - converting to formats, 180–181
 - converting to frequency dictionaries, 187
 - converting to numbers, 23
 - creating, 19
 - defined, 17
 - empty, 18
 - escape characters, 88
 - extracting substrings from, 89
 - formatting, 124–127
 - immutable quality, 28
 - indexing, 84–86
 - indicating, 17
 - inserting at start of files, 137
 - lengths, 18
 - number of characters in, 18
 - printing on screen, 39–40
 - reading from keyboard, 36–38
 - regular expressions, 98–100
 - representations of objects, 159
 - returning list of, 131
 - slicing, 89–91
 - splitting, 178–179
 - square brackets ([]) for indexing, 84
 - uses of, 9
 - using quotes (' and ") with, 17
 - using **strip()** function with, 37
 - as words, 179
- string-searching functions
 - s.find()**, 93
 - s.index()**, 93
 - s.rfind()**, 93
 - s.rindex()**, 93
- string-splitting functions
 - s.partition()**, 95
 - s.rpartition()**, 95
 - s.rsplit()**, 95
 - s.split()**, 95
 - s.splitlines()**, 95
- string-stripping functions
 - s.lstrip()**, 95
 - s.rstrip()**, 95
 - s.strip()**, 95

- string-testing functions
 - for contents of substrings, 92
 - `s.endswith()`, 92
 - `s.isalnum()`, 92
 - `s.isalpha()`, 92
 - `s.isdecimal()`, 92
 - `s.isdigit()`, 92
 - `s.isidentifier()`, 92
 - `s.islower()`, 92
 - `s.isnumeric()`, 92
 - `s.isprintable()`, 92
 - `s.isspace()`, 92
 - `s.istitle()`, 92
 - `s.isupper()`, 92
 - `s.startswith()`, 92
- `strip()` function, using with strings, 37
- subclasses, using with classes, 169–170
- substrings, extracting from strings, 89
- subtraction (-) operator, 12
- summing
 - numbers, 62
 - numbers from users, 61
- syntax errors, causing, 145

T

- 't file module, meaning of, 134, 137
- `tan(x)` function, 16
- templating packages, using, 126
- testing functions. *See* Boolean logic; string-testing functions
- text files
 - appending to, 136
 - closing, 134
 - opening, 134
 - processing, 134–137
 - reading as strings, 135
 - reading line by line, 134–137
 - writing to, 136
- text mode, indicating, 134
- text statistics case study
 - completing, 188–189
 - converting strings to formats, 180–181
 - final program, 192–193
 - finding frequent words, 184–186
 - `normalize()` function, 180–181
 - problem description, 178–179

- regular expressions, 181
- strings to frequency dictionary, 187
- testing code on data file, 182–183
- text vs. binary files, 128–129
- `this` module, importing at command line, 82
- Tkinter package, 196
- tracebacks, outputting, 144
- tracing programs, 36–37
- transcripts, explained, 10
- `True` values, returning for paths, 131
- try/except blocks
 - adding `finally` code block to, 150
 - examples of, 146–148
 - in Undercut game, 172
- tuple functions. *See also* functions
 - `len()`, 106
 - `tup.count()`, 106
 - `tup.index()`, 106
 - `x in tup`, 106
- tuples. *See also* lists
 - concatenating, 107
 - creating list of, 185–186
 - defined, 103
 - example of, 95
 - immutability, 105
 - round brackets (`()`), 104
 - singleton, 104
 - sorting, 114
 - trailing commas, 104
 - writing values as, 29
- Twisted network programming package, 197
- `type` command, using, 102
- types. *See* data types

U

- Undercut game
 - implementing, 171–174
 - `move` functions, 172–173
 - playing, 173–174
 - try/except blocks, 172
- Unicode, rise of, 87
- uppercase float exponential, conversion
 - specifier for, 125
- uppercase hexadecimal, conversion specifier
 - for, 125
- `urllib` module, using, 141

V

ValueError example, 146–147

values. *See also* sequences

- assigning in parallel, 30
- assigning to variables, 27
- displaying multiple, 29
- referring variables to, 28
- replacing by position, 126
- and variables, 24–25
- writing as tuples, 29

variable names

- case sensitivity, 25
- first character, 25
- keywords, 25
- lengths, 25
- rules for, 25

variable scope

- explained, 73
- global variables, 74
- local variables, 73

variable values, swapping, 30

variables

- adding multiple, 29
- assigned values, 27
- assigning values to, 27
- explained, 9
- pointing to values, 27
- private vs. public, 166–167
- referring to values, 28
- terminology, 27
- and values, 24–25

virtual machine, explained, 35

von Rossum, Guido, 2

W

'w' file module, meaning of, 134

web browsers, creating, 141

webbrowser module, explained, 141

webpages, reading, 141

websites

- 2to3 conversion for Python, 201
- Bottle, 196
- Django, 196
- online documentation, 133

PIL (Python Imaging Library), 196

Pygame, 197

PyPI (Python Package Index), 197

Python download page, 6

pythonintro, 133

re module documentation, 100

SciPy, 197

templating packages, 126

Tkinter, 196

Twisted, 197

Unicode home page, 87

while-loops

- flexibility of, 58
- flow of control, 56
- vs. for-loops, 58–63
- form of, 57
- incrementers, 57
- initializers, 57
- sample program, 56
- try/except block in, 146

whitespace characters, handling, 88

Windows, installing Python on, 6

with statement, using, 151

word counts, preceding with brackets, 186

words

- creating set of stop words, 190
- finding frequent, 184–186
- getting sorted list of, 185–186
- splitting sentences into, 179
- strings as, 179

writing

- data structures, 139
- files, 128–130
- opening text files for, 134
- to text files, 136

X

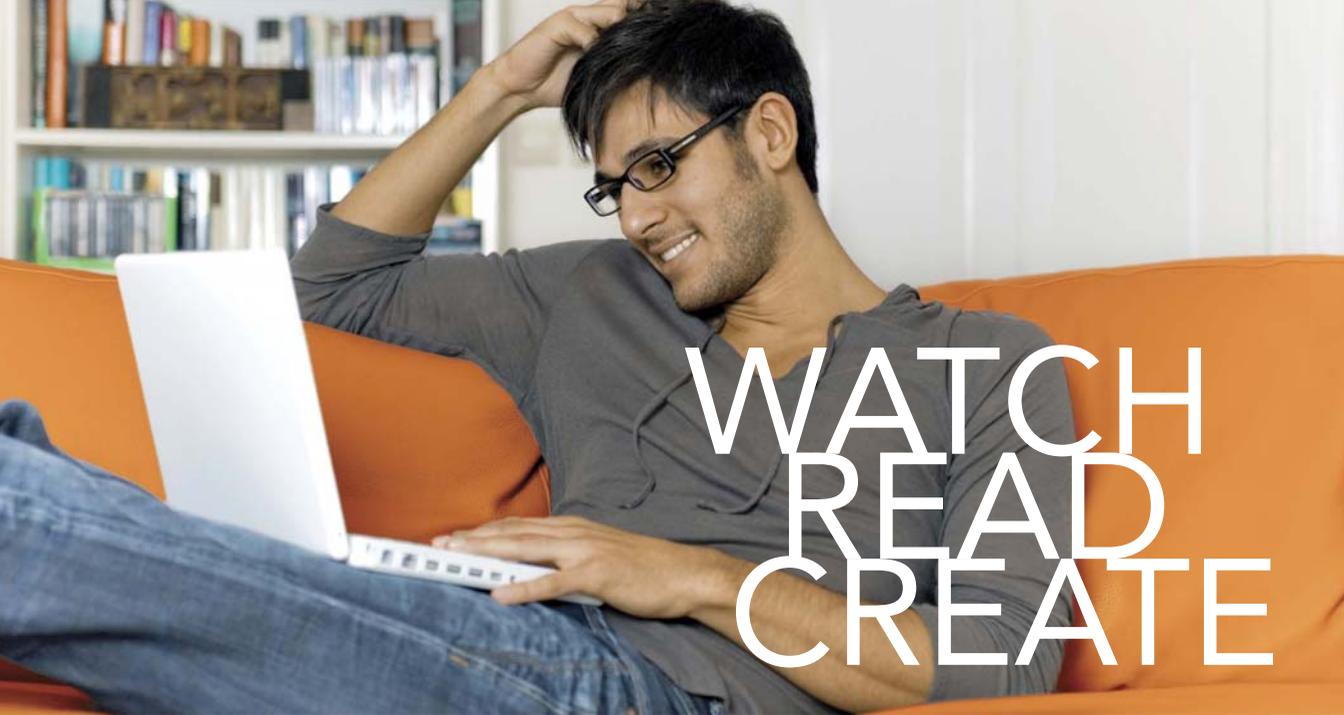
x = expr, 28

x conversion specifier, meaning of, 125

X conversion specifier, meaning of, 125

Z

zfill function, using, 97



WATCH READ CREATE

Unlimited online access to all Peachpit, Adobe Press, Apple Training and New Riders videos and books, as well as content from other leading publishers including: O'Reilly Media, Focal Press, Sams, Que, Total Training, John Wiley & Sons, Course Technology PTR, Class on Demand, VTC and more.

No time commitment or contract required!
Sign up for one month or a year.
All for \$19.99 a month

SIGN UP TODAY
peachpit.com/creativeedge

creative
edge