كليه الحاسبات والمعلومات

Fundamentals of

Python

Programming for Beginners

CONTENTS IN DETAIL

INTRODUCTION 1

Whom Is This Book For? 2
Conventions 2
What Is Programming? 3
    What Is Python? 4
    Programmers Don't Need to Know Much Math 4
    Programming Is a Creative Activity 5
About This Book 5
Downloading and Installing Python 6
Starting IDLE 7
    The Interactive Shell 8
How to Find Help 8
Asking Smart Programming Questions 9
Summary 10

PART I: PYTHON PROGRAMMING BASICS 11

1
PYTHON BASICS 13

Entering Expressions into the Interactive Shell 14
The Integer, Floating-Point, and String Data Types 16
String Concatenation and Replication 17
Storing Values in Variables 18
    Assignment Statements 18
    Variable Names 20
Your First Program 21
Dissecting Your Program 22
    Comments 23
    The print() Function 23
    The input() Function 23
    Printing the User's Name 24
    The len() Function 24
    The str(), int(), and float() Functions 25
Summary 28
Practice Questions 28

2
FLOW CONTROL 31

Boolean Values 32
Comparison Operators 33
Boolean Operators 35

Binary Boolean Operators . . . 35
The not Operator . . . 36
Mixing Boolean and Comparison Operators . . . 36
Elements of Flow Control . . . 37
Conditions . . . 37
Blocks of Code . . . 37
Program Execution . . . 38
Flow Control Statements . . . 38
if Statements . . . 38
else Statements . . . 39
elif Statements . . . 40
while Loop Statements . . . 45
break Statements . . . 49
continue Statements . . . 50
for Loops and the range() Function . . . 53
Importing Modules . . . 57
from import Statements . . . 58
Ending a Program Early with sys.exit() . . . 58
Summary . . . 58
Practice Questions . . . 59

3
FUNCTIONS 61

def Statements with Parameters . . . 63
Return Values and return Statements . . . 63
The None Value . . . 65
Keyword Arguments and print() . . . 65
Local and Global Scope . . . 67
Local Variables Cannot Be Used in the Global Scope . . . 67
Local Scopes Cannot Use Variables in Other Local Scopes . . . 68
Global Variables Can Be Read from a Local Scope . . . 69
Local and Global Variables with the Same Name . . . 69
The global Statement . . . 70
Exception Handling . . . 72
A Short Program: Guess the Number . . . 74
Summary . . . 76
Practice Questions . . . 76
Practice Projects . . . 77
The Collatz Sequence . . . 77
Input Validation . . . 77

4
LISTS 79

The List Data Type . . . 80
Getting Individual Values in a List with Indexes . . . 80
Negative Indexes . . . 82
Getting Sublists with Slices . . . 82
Getting a List's Length with len() . . . 83
Changing Values in a List with Indexes . . . 83

List Concatenation and List Replication 83
Removing Values from Lists with del Statements 84
Working with Lists 84
Using for Loops with Lists 86
The in and not in Operators 87
The Multiple Assignment Trick 87
Augmented Assignment Operators 88
Methods 89
Finding a Value in a List with the index() Method 89
Adding Values to Lists with the append() and insert() Methods 89
Removing Values from Lists with remove() 90
Sorting the Values in a List with the sort() Method 91
Example Program: Magic 8 Ball with a List 92
List-like Types: Strings and Tuples 93
Mutable and Immutable Data Types 94
The Tuple Data Type 96
Converting Types with the list() and tuple() Functions 97
References 97
Passing References 100
The copy Module's copy() and deepcopy() Functions 100
Summary 101
Practice Questions 102
Practice Projects 102
Comma Code 102
Character Picture Grid 103

5
DICTIONARIES AND STRUCTURING DATA 105

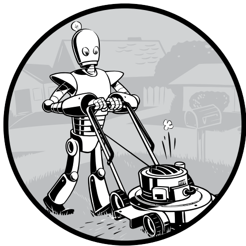
The Dictionary Data Type 105
Dictionaries vs. Lists 106
The keys(), values(), and items() Methods 107
Checking Whether a Key or Value Exists in a Dictionary 109
The get() Method 109
The setdefault() Method 110
Pretty Printing 111
Using Data Structures to Model Real-World Things 112
A Tic-Tac-Toe Board 113
Nested Dictionaries and Lists 117
Summary 119
Practice Questions 119
Practice Projects 120
Fantasy Game Inventory 120
List to Dictionary Function for Fantasy Game Inventory 120

6
MANIPULATING STRINGS 123

Working with Strings 123
String Literals 124
Indexing and Slicing Strings 126
The in and not in Operators with Strings 127

Useful String Methods 127
The upper(), lower(), isupper(), and islower() String Methods 128
The isX String Methods 129
The startswith() and endswith() String Methods 131
The join() and split() String Methods 131
Justifying Text with rjust(), ljust(), and center() 133
Removing Whitespace with strip(), rstrip(), and lstrip() 134
Copying and Pasting Strings with the pyperclip Module 135
Project: Password Locker 136
Step 1: Program Design and Data Structures 136
Step 2: Handle Command Line Arguments 137
Step 3: Copy the Right Password 137
Project: Adding Bullets to Wiki Markup 139
Step 1: Copy and Paste from the Clipboard 139
Step 2: Separate the Lines of Text and Add the Star 140
Step 3: Join the Modified Lines 141
Summary 141
Practice Questions 142
Practice Project 142
Table Printer 142

INTRODUCTION



"You've just done in two hours what it takes the three of us two days to do." My college roommate was working at a retail electronics store in the early 2000s. Occasionally, the store would receive a spreadsheet of thousands of product prices from its competitor. A team of three employees would print the spreadsheet onto a thick stack of paper and split it among themselves. For each product price, they would look up their store's price and note all the products that their competitors sold for less. It usually took a couple of days.

"You know, I could write a program to do that if you have the original file for the printouts," my roommate told them, when he saw them sitting on the floor with papers scattered and stacked around them.

After a couple of hours, he had a short program that read a competitor's price from a file, found the product in the store's database, and noted whether the competitor was cheaper. He was still new to programming, and

he spent most of his time looking up documentation in a programming book. The actual program took only a few seconds to run. My roommate and his co-workers took an extra-long lunch that day.

This is the power of computer programming. A computer is like a Swiss Army knife that you can configure for countless tasks. Many people spend hours clicking and typing to perform repetitive tasks, unaware that the machine they're using could do their job in seconds if they gave it the right instructions.

Whom Is This Book For?

Software is at the core of so many of the tools we use today: Nearly everyone uses social networks to communicate, many people have Internet-connected computers in their phones, and most office jobs involve interacting with a computer to get work done. As a result, the demand for people who can code has skyrocketed. Countless books, interactive web tutorials, and developer boot camps promise to turn ambitious beginners into software engineers with six-figure salaries.

This book is not for those people. It's for everyone else.

On its own, this book won't turn you into a professional software developer any more than a few guitar lessons will turn you into a rock star. But if you're an office worker, administrator, academic, or anyone else who uses a computer for work or fun, you will learn the basics of programming so that you can automate simple tasks such as the following:

- Moving and renaming thousands of files and sorting them into folders
- Filling out online forms, no typing required
- Downloading files or copy text from a website whenever it updates
- Having your computer text you custom notifications
- Updating or formatting Excel spreadsheets
- Checking your email and sending out prewritten responses

These tasks are simple but time-consuming for humans, and they're often so trivial or specific that there's no ready-made software to perform them. Armed with a little bit of programming knowledge, you can have your computer do these tasks for you.

Conventions

This book is not designed as a reference manual; it's a guide for beginners. The coding style sometimes goes against best practices (for example, some programs use global variables), but that's a trade-off to make the code simpler to learn. This book is made for people to write throwaway code, so there's not much time spent on style and elegance. Sophisticated programming concepts—like object-oriented programming, list comprehensions,

and generators—aren't covered because of the complexity they add. Veteran programmers may point out ways the code in this book could be changed to improve efficiency, but this book is mostly concerned with getting programs to work with the least amount of effort.

What Is Programming?

Television shows and films often show programmers furiously typing cryptic streams of 1s and 0s on glowing screens, but modern programming isn't that mysterious. *Programming* is simply the act of entering instructions for the computer to perform. These instructions might crunch some numbers, modify text, look up information in files, or communicate with other computers over the Internet.

All programs use basic instructions as building blocks. Here are a few of the most common ones, in English:

"Do this; then do that."

"If this condition is true, perform this action; otherwise, do that action."

"Do this action that number of times."

"Keep doing that until this condition is true."

You can combine these building blocks to implement more intricate decisions, too. For example, here are the programming instructions, called the *source code*, for a simple program written in the Python programming language. Starting at the top, the Python software runs each line of code (some lines are run only *if* a certain condition is true or *else* Python runs some other line) until it reaches the bottom.

```
❶ passwordFile = open('SecretPasswordFile.txt')
❷ secretPassword = passwordFile.read()
❸ print('Enter your password.')
  typedPassword = input()
❹ if typedPassword == secretPassword:
❺    print('Access granted')
❻    if typedPassword == '12345':
❼       print('That password is one that an idiot puts on their luggage.')
  else:
❽    print('Access denied')
```

You might not know anything about programming, but you could probably make a reasonable guess at what the previous code does just by reading it. First, the file *SecretPasswordFile.txt* is opened ❶, and the secret password in it is read ❷. Then, the user is prompted to input a password (from the keyboard) ❸. These two passwords are compared ❹, and if they're the same, the program prints *Access granted* to the screen ❺. Next, the program checks to see whether the password is *12345* ❻ and hints that this choice might not be the best for a password ❼. If the passwords are not the same, the program prints *Access denied* to the screen ❽.

What Is Python?

Python refers to the Python programming language (with syntax rules for writing what is considered valid Python code) and the Python interpreter software that reads source code (written in the Python language) and performs its instructions. The Python interpreter is free to download from *http://python.org/*, and there are versions for Linux, OS X, and Windows.

The name Python comes from the surreal British comedy group Monty Python, not from the snake. Python programmers are affectionately called Pythonistas, and both Monty Python and serpentine references usually pepper Python tutorials and documentation.

Programmers Don't Need to Know Much Math

The most common anxiety I hear about learning to program is that people think it requires a lot of math. Actually, most programming doesn't require math beyond basic arithmetic. In fact, being good at programming isn't that different from being good at solving Sudoku puzzles.

To solve a Sudoku puzzle, the numbers 1 through 9 must be filled in for each row, each column, and each 3×3 interior square of the full 9×9 board. You find a solution by applying deduction and logic from the starting numbers. For example, since 5 appears in the top left of the Sudoku puzzle shown in Figure 0-1, it cannot appear elsewhere in the top row, in the leftmost column, or in the top-left 3×3 square. Solving one row, column, or square at a time will provide more number clues for the rest of the puzzle.

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

Figure 0-1: A new Sudoku puzzle (left) and its solution (right). Despite using numbers, Sudoku doesn't involve much math. (Images © Wikimedia Commons)

Just because Sudoku involves numbers doesn't mean you have to be good at math to figure out the solution. The same is true of programming. Like solving a Sudoku puzzle, writing programs involves breaking down a problem into individual, detailed steps. Similarly, when *debugging* programs (that is, finding and fixing errors), you'll patiently observe what the program is doing and find the cause of the bugs. And like all skills, the more you program, the better you'll become.

Programming Is a Creative Activity

Programming is a creative task, somewhat like constructing a castle out of LEGO bricks. You start with a basic idea of what you want your castle to look like and inventory your available blocks. Then you start building. Once you've finished building your program, you can pretty up your code just like you would your castle.

The difference between programming and other creative activities is that when programming, you have all the raw materials you need in your computer; you don't need to buy any additional canvas, paint, film, yarn, LEGO bricks, or electronic components. When your program is written, it can easily be shared online with the entire world. And though you'll make mistakes when programming, the activity is still a lot of fun.

About This Book

The first part of this book covers basic Python programming concepts, and the second part covers various tasks you can have your computer automate. Each chapter in the second part has project programs for you to study. Here's a brief rundown of what you'll find in each chapter:

Part I: Python Programming Basics

Chapter 1: Python Basics Covers expressions, the most basic type of Python instruction, and how to use the Python interactive shell software to experiment with code.

Chapter 2: Flow Control Explains how to make programs decide which instructions to execute so your code can intelligently respond to different conditions.

Chapter 3: Functions Instructs you on how to define your own functions so that you can organize your code into more manageable chunks.

Chapter 4: Lists Introduces the list data type and explains how to organize data.

Chapter 5: Dictionaries and Structuring Data Introduces the dictionary data type and shows you more powerful ways to organize data.

Chapter 6: Manipulating Strings Covers working with text data (called *strings* in Python).

Part II: Automating Tasks

Chapter 7: Pattern Matching with Regular Expressions Covers how Python can manipulate strings and search for text patterns with regular expressions.

Chapter 8: Reading and Writing Files Explains how your programs can read the contents of text files and save information to files on your hard drive.

Chapter 9: Organizing Files Shows how Python can copy, move, rename, and delete large numbers of files much faster than a human user can. It also explains compressing and decompressing files.

Chapter 10: Debugging Shows how to use Python's various bug-finding and bug-fixing tools.

Chapter 11: Web Scraping Shows how to write programs that can automatically download web pages and parse them for information. This is called *web scraping*.

Chapter 12: Working with Excel Spreadsheets Covers programmatically manipulating Excel spreadsheets so that you don't have to read them. This is helpful when the number of documents you have to analyze is in the hundreds or thousands.

Chapter 13: Working with PDF and Word Documents Covers programmatically reading Word and PDF documents.

Chapter 14: Working with CSV Files and JSON Data Continues to explain how to programmatically manipulate documents with CSV and JSON files.

Chapter 15: Keeping Time, Scheduling Tasks, and Launching Programs Explains how time and dates are handled by Python programs and how to schedule your computer to perform tasks at certain times. This chapter also shows how your Python programs can launch non-Python programs.

Chapter 16: Sending Email and Text Messages Explains how to write programs that can send emails and text messages on your behalf.

Chapter 17: Manipulating Images Explains how to programmatically manipulate images such as JPEG or PNG files.

Chapter 18: Controlling the Keyboard and Mouse with GUI Automation Explains how to programmatically control the mouse and keyboard to automate clicks and keypresses.

Downloading and Installing Python

You can download Python for Windows, OS X, and Ubuntu for free from *http://python.org/downloads/*. If you download the latest version from the website's download page, all of the programs in this book should work.

WARNING *Be sure to download a version of Python 3 (such as 3.4.0). The programs in this book are written to run on Python 3 and may not run correctly, if at all, on Python 2.*

You'll find Python installers for 64-bit and 32-bit computers for each operating system on the download page, so first figure out which installer you need. If you bought your computer in 2007 or later, it is most likely a 64-bit system. Otherwise, you have a 32-bit version, but here's how to find out for sure:

- On Windows, select **Start ▸ Control Panel ▸ System** and check whether System Type says 64-bit or 32-bit.

- On OS X, go the Apple menu, select **About This Mac ▸ More Info ▸ System Report ▸ Hardware**, and then look at the Processor Name field. If it says Intel Core Solo or Intel Core Duo, you have a 32-bit machine. If it says anything else (including Intel Core 2 Duo), you have a 64-bit machine.
- On Ubuntu Linux, open a Terminal and run the command `uname -m`. A response of `i686` means 32-bit, and `x86_64` means 64-bit.

On Windows, download the Python installer (the filename will end with *.msi*) and double-click it. Follow the instructions the installer displays on the screen to install Python, as listed here:

1. Select **Install for All Users** and then click **Next**.
2. Install to the *C:\Python34* folder by clicking **Next**.
3. Click **Next** again to skip the Customize Python section.

On Mac OS X, download the *.dmg* file that's right for your version of OS X and double-click it. Follow the instructions the installer displays on the screen to install Python, as listed here:

1. When the DMG package opens in a new window, double-click the *Python.mpkg* file. You may have to enter the administrator password.
2. Click **Continue** through the Welcome section and click **Agree** to accept the license.
3. Select **HD Macintosh** (or whatever name your hard drive has) and click **Install**.

If you're running Ubuntu, you can install Python from the Terminal by following these steps:

1. Open the Terminal window.
2. Enter **`sudo apt-get install python3`**.
3. Enter **`sudo apt-get install idle3`**.
4. Enter **`sudo apt-get install python3-pip`**.

Starting IDLE

While the *Python interpreter* is the software that runs your Python programs, the *interactive development environment (IDLE)* software is where you'll enter your programs, much like a word processor. Let's start IDLE now.

- On Windows 7 or newer, click the Start icon in the lower-left corner of your screen, enter **`IDLE`** in the search box, and select **IDLE (Python GUI)**.
- On Windows XP, click the **Start** button and then select **Programs ▸ Python 3.4 ▸ IDLE (Python GUI)**.

- On Mac OS X, open the Finder window, click **Applications**, click **Python 3.4**, and then click the IDLE icon.
- On Ubuntu, select **Applications** ▸ **Accessories** ▸ **Terminal** and then enter **`idle3`**. (You may also be able to click **Applications** at the top of the screen, select **Programming**, and then click **IDLE 3**.)

The Interactive Shell

No matter which operating system you're running, the IDLE window that first appears should be mostly blank except for text that looks something like this:

```
Python 3.4.0 (v3.4.0:04f714765c13, Mar 16 2014, 19:25:23) [MSC v.1600 64
bit (AMD64)] on win32Type "copyright", "credits" or "license()" for more
information.
>>>
```

This window is called the *interactive shell*. A shell is a program that lets you type instructions into the computer, much like the Terminal or Command Prompt on OS X and Windows, respectively. Python's interactive shell lets you enter instructions for the Python interpreter software to run. The computer reads the instructions you enter and runs them immediately.

For example, enter the following into the interactive shell next to the >>> prompt:

```
>>> print('Hello world!')
```

After you type that line and press ENTER, the interactive shell should display this in response:

```
>>> print('Hello world!')
Hello world!
```

How to Find Help

Solving programming problems on your own is easier than you might think. If you're not convinced, then let's cause an error on purpose: Enter **`'42' + 3`** into the interactive shell. You don't need to know what this instruction means right now, but the result should look like this:

```
  >>> '42' + 3
❶ Traceback (most recent call last):
    File "<pyshell#0>", line 1, in <module>
      '42' + 3
❷ TypeError: Can't convert 'int' object to str implicitly
  >>>
```

The error message ❷ appeared here because Python couldn't understand your instruction. The traceback part ❶ of the error message shows the specific instruction and line number that Python had trouble with. If you're not sure what to make of a particular error message, search online for the exact error message. Enter **"TypeError: Can't convert 'int' object to str implicitly"** (including the quotes) into your favorite search engine, and you should see tons of links explaining what the error message means and what causes it, as shown in Figure 0-2.

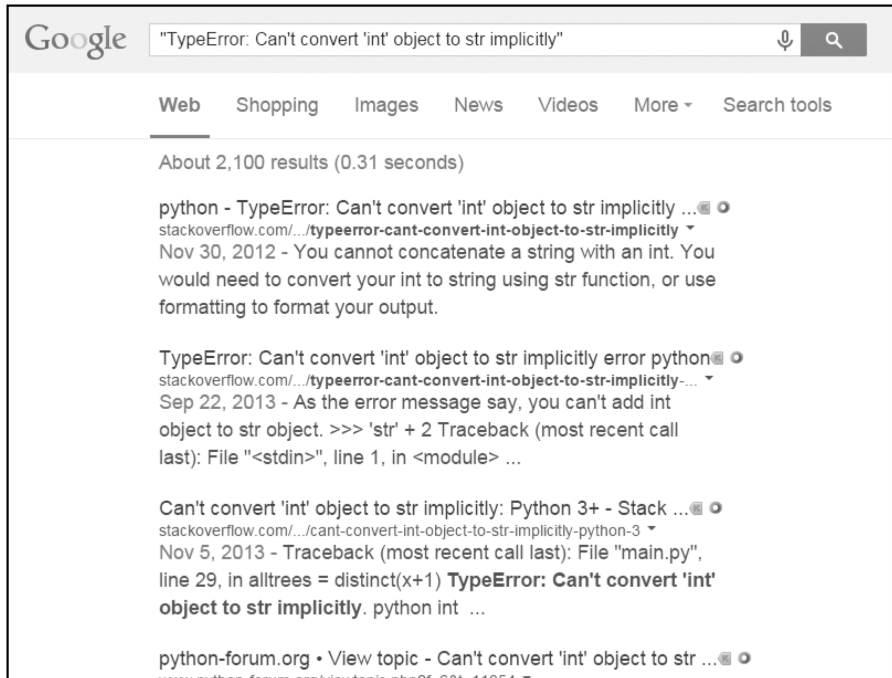


Figure 0-2: The Google results for an error message can be very helpful.

You'll often find that someone else had the same question as you and that some other helpful person has already answered it. No one person can know everything about programming, so an everyday part of any software developer's job is looking up answers to technical questions.

Asking Smart Programming Questions

If you can't find the answer by searching online, try asking people in a web forum such as Stack Overlow (*http://stackoverflow.com/*) or the "learn programming" subreddit at *http://reddit.com/r/learnprogramming/*. But keep in mind there are smart ways to ask programming questions that help others help you. Be sure to read the Frequently Asked Questions sections these websites have about the proper way to post questions.

When asking programming questions, remember to do the following:

- Explain what you are trying to do, not just what you did. This lets your helper know if you are on the wrong track.
- Specify the point at which the error happens. Does it occur at the very start of the program or only after you do a certain action?
- Copy and paste the *entire* error message and your code to *http://pastebin.com/* or *http://gist.github.com/*.

  These websites make it easy to share large amounts of code with people over the Web, without the risk of losing any text formatting. You can then put the URL of the posted code in your email or forum post. For example, here some pieces of code I've posted: *http://pastebin.com/SzP2DbFx/* and *https://gist.github.com/asweigart/6912168/*.
- Explain what you've already tried to do to solve your problem. This tells people you've already put in some work to figure things out on your own.
- List the version of Python you're using. (There are some key differences between version 2 Python interpreters and version 3 Python interpreters.) Also, say which operating system and version you're running.
- If the error came up after you made a change to your code, explain exactly what you changed.
- Say whether you're able to reproduce the error every time you run the program or whether it happens only after you perform certain actions. Explain what those actions are, if so.

Always follow good online etiquette as well. For example, don't post your questions in all caps or make unreasonable demands of the people trying to help you.

Summary

For most people, their computer is just an appliance instead of a tool. But by learning how to program, you'll gain access to one of the most powerful tools of the modern world, and you'll have fun along the way. Programming isn't brain surgery—it's fine for amateurs to experiment and make mistakes.

I love helping people discover Python. I write programming tutorials on my blog at *http://inventwithpython.com/blog/*, and you can contact me with questions at *al@inventwithpython.com*.

This book will start you off from zero programming knowledge, but you may have questions beyond its scope. Remember that asking effective questions and knowing how to find answers are invaluable tools on your programming journey.

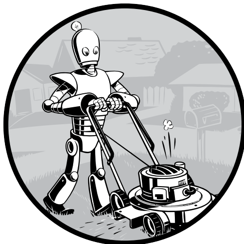
Let's begin!

PART I

PYTHON PROGRAMMING BASICS

1

PYTHON BASICS



The Python programming language has a wide range of syntactical constructions, standard library functions, and interactive development environment features. Fortunately, you can ignore most of that; you just need to learn enough to write some handy little programs.

You will, however, have to learn some basic programming concepts before you can do anything. Like a wizard-in-training, you might think these concepts seem arcane and tedious, but with some knowledge and practice, you'll be able to command your computer like a magic wand to perform incredible feats.

This chapter has a few examples that encourage you to type into the interactive shell, which lets you execute Python instructions one at a time and shows you the results instantly. Using the interactive shell is great for learning what basic Python instructions do, so give it a try as you follow along. You'll remember the things you do much better than the things you only read.

Entering Expressions into the Interactive Shell

You run the interactive shell by launching IDLE, which you installed with Python in the introduction. On Windows, open the Start menu, select **All Programs ▸ Python 3.3**, and then select **IDLE (Python GUI)**. On OS X, select **Applications ▸ MacPython 3.3 ▸ IDLE**. On Ubuntu, open a new Terminal window and enter **idle3**.

A window with the >>> prompt should appear; that's the interactive shell. Enter **2 + 2** at the prompt to have Python do some simple math.

```
>>> 2 + 2
4
```

The IDLE window should now show some text like this:

```
Python 3.3.2 (v3.3.2:d047928ae3f6, May 16 2013, 00:06:53) [MSC v.1600 64 bit
(AMD64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> 2 + 2
4
>>>
```

In Python, `2 + 2` is called an *expression*, which is the most basic kind of programming instruction in the language. Expressions consist of *values* (such as `2`) and *operators* (such as `+`), and they can always *evaluate* (that is, reduce) down to a single value. That means you can use expressions anywhere in Python code that you could also use a value.

In the previous example, `2 + 2` is evaluated down to a single value, `4`. A single value with no operators is also considered an expression, though it evaluates only to itself, as shown here:

```
>>> 2
2
```

ERRORS ARE OKAY!

Programs will crash if they contain code the computer can't understand, which will cause Python to show an error message. An error message won't break your computer, though, so don't be afraid to make mistakes. A *crash* just means the program stopped running unexpectedly.

If you want to know more about an error message, you can search for the exact message text online to find out more about that specific error. You can also check out the resources at *http://nostarch.com/automatestuff/* to see a list of common Python error messages and their meanings.

There are plenty of other operators you can use in Python expressions, too. For example, Table 1-1 lists all the math operators in Python.

Table 1-1: Math Operators from Highest to Lowest Precedence

Operator	Operation	Example	Evaluates to...
**	Exponent	2 ** 3	8
%	Modulus/remainder	22 % 8	6
//	Integer division/floored quotient	22 // 8	2
/	Division	22 / 8	2.75
*	Multiplication	3 * 5	15
-	Subtraction	5 - 2	3
+	Addition	2 + 2	4

The order of operations (also called *precedence*) of Python math operators is similar to that of mathematics. The ** operator is evaluated first; the *, /, //, and % operators are evaluated next, from left to right; and the + and - operators are evaluated last (also from left to right). You can use parentheses to override the usual precedence if you need to. Enter the following expressions into the interactive shell:

```
>>> 2 + 3 * 6
20
>>> (2 + 3) * 6
30
>>> 48565878 * 578453
28093077826734
>>> 2 ** 8
256
>>> 23 / 7
3.2857142857142856
>>> 23 // 7
3
>>> 23 % 7
2
>>> 2      +            2
4
>>> (5 - 1) * ((7 + 1) / (3 - 1))
16.0
```

In each case, you as the programmer must enter the expression, but Python does the hard part of evaluating it down to a single value. Python will keep evaluating parts of the expression until it becomes a single value, as shown in Figure 1-1.

```
(5 - 1) * ((7 + 1) / (3 - 1))
        ↓
      4 * ((7 + 1) / (3 - 1))
        ↓
      4 * (   8   ) / (3 - 1))
        ↓
      4 * (   8   ) / (   2   )
        ↓
      4 * 4.0
        ↓
       16.0
```

Figure 1-1: Evaluating an expression reduces it to a single value.

These rules for putting operators and values together to form expressions are a fundamental part of Python as a programming language, just like the grammar rules that help us communicate. Here's an example:

This is a grammatically correct English sentence.

This grammatically is sentence not English correct a.

The second line is difficult to parse because it doesn't follow the rules of English. Similarly, if you type in a bad Python instruction, Python won't be able to understand it and will display a `SyntaxError` error message, as shown here:

```
>>> 5 +
  File "<stdin>", line 1
    5 +
      ^
SyntaxError: invalid syntax
>>> 42 + 5 + * 2
  File "<stdin>", line 1
    42 + 5 + * 2
             ^
SyntaxError: invalid syntax
```

You can always test to see whether an instruction works by typing it into the interactive shell. Don't worry about breaking the computer: The worst thing that could happen is that Python responds with an error message. Professional software developers get error messages while writing code all the time.

The Integer, Floating-Point, and String Data Types

Remember that expressions are just values combined with operators, and they always evaluate down to a single value. A *data type* is a category for values, and every value belongs to exactly one data type. The most

common data types in Python are listed in Table 1-2. The values `-2` and `30`, for example, are said to be *integer* values. The integer (or *int*) data type indicates values that are whole numbers. Numbers with a decimal point, such as `3.14`, are called *floating-point numbers* (or *floats*). Note that even though the value `42` is an integer, the value `42.0` would be a floating-point number.

Table 1-2: Common Data Types

Data type	Examples
Integers	`-2, -1, 0, 1, 2, 3, 4, 5`
Floating-point numbers	`-1.25, -1.0, --0.5, 0.0, 0.5, 1.0, 1.25`
Strings	`'a', 'aa', 'aaa', 'Hello!', '11 cats'`

Python programs can also have text values called *strings*, or *strs* (pronounced "stirs"). Always surround your string in single quote (`'`) characters (as in `'Hello'` or `'Goodbye cruel world!'`) so Python knows where the string begins and ends. You can even have a string with no characters in it, `''`, called a *blank string*. Strings are explained in greater detail in Chapter 4.

If you ever see the error message `SyntaxError: EOL while scanning string literal`, you probably forgot the final single quote character at the end of the string, such as in this example:

```
>>> 'Hello world!
SyntaxError: EOL while scanning string literal
```

String Concatenation and Replication

The meaning of an operator may change based on the data types of the values next to it. For example, `+` is the addition operator when it operates on two integers or floating-point values. However, when `+` is used on two string values, it joins the strings as the *string concatenation* operator. Enter the following into the interactive shell:

```
>>> 'Alice' + 'Bob'
'AliceBob'
```

The expression evaluates down to a single, new string value that combines the text of the two strings. However, if you try to use the `+` operator on a string and an integer value, Python will not know how to handle this, and it will display an error message.

```
>>> 'Alice' + 42
Traceback (most recent call last):
  File "<pyshell#26>", line 1, in <module>
    'Alice' + 42
TypeError: Can't convert 'int' object to str implicitly
```

The error message `Can't convert 'int' object to str implicitly` means that Python thought you were trying to concatenate an integer to the string `'Alice'`. Your code will have to explicitly convert the integer to a string, because Python cannot do this automatically. (Converting data types will be explained in "Dissecting Your Program" on page 22 when talking about the `str()`, `int()`, and `float()` functions.)

The * operator is used for multiplication when it operates on two integer or floating-point values. But when the * operator is used on one string value and one integer value, it becomes the *string replication* operator. Enter a string multiplied by a number into the interactive shell to see this in action.

```
>>> 'Alice' * 5
'AliceAliceAliceAliceAlice'
```

The expression evaluates down to a single string value that repeats the original a number of times equal to the integer value. String replication is a useful trick, but it's not used as often as string concatenation.

The * operator can be used with only two numeric values (for multiplication) or one string value and one integer value (for string replication). Otherwise, Python will just display an error message.

```
>>> 'Alice' * 'Bob'
Traceback (most recent call last):
  File "<pyshell#32>", line 1, in <module>
    'Alice' * 'Bob'
TypeError: can't multiply sequence by non-int of type 'str'
>>> 'Alice' * 5.0
Traceback (most recent call last):
  File "<pyshell#33>", line 1, in <module>
    'Alice' * 5.0
TypeError: can't multiply sequence by non-int of type 'float'
```

It makes sense that Python wouldn't understand these expressions: You can't multiply two words, and it's hard to replicate an arbitrary string a fractional number of times.

Storing Values in Variables

A *variable* is like a box in the computer's memory where you can store a single value. If you want to use the result of an evaluated expression later in your program, you can save it inside a variable.

Assignment Statements

You'll store values in variables with an *assignment statement*. An assignment statement consists of a variable name, an equal sign (called the *assignment operator*), and the value to be stored. If you enter the assignment statement `spam = 42`, then a variable named `spam` will have the integer value `42` stored in it.

Think of a variable as a labeled box that a value is placed in, as in Figure 1-2.

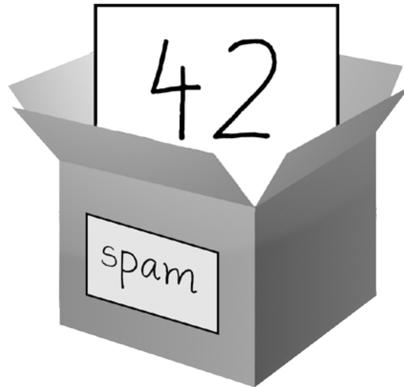


Figure 1-2: `spam = 42` *is like telling the program, "The variable* `spam` *now has the integer value* `42` *in it."*

For example, enter the following into the interactive shell:

```
❶ >>> spam = 40
  >>> spam
  40
  >>> eggs = 2
❷ >>> spam + eggs
  42
  >>> spam + eggs + spam
  82
❸ >>> spam = spam + 2
  >>> spam
  42
```

A variable is *initialized* (or created) the first time a value is stored in it ❶. After that, you can use it in expressions with other variables and values ❷. When a variable is assigned a new value ❸, the old value is forgotten, which is why `spam` evaluated to `42` instead of `40` at the end of the example. This is called *overwriting* the variable. Enter the following code into the interactive shell to try overwriting a string:

```
>>> spam = 'Hello'
>>> spam
'Hello'
>>> spam = 'Goodbye'
>>> spam
'Goodbye'
```

Just like the box in Figure 1-3, the `spam` variable in this example stores `'Hello'` until you replace it with `'Goodbye'`.

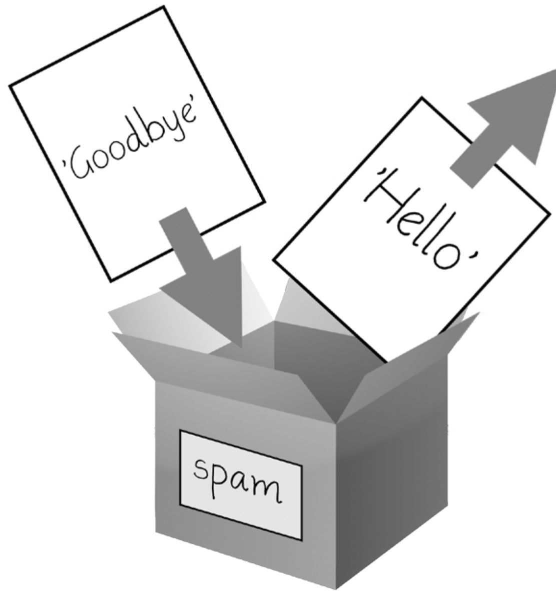


Figure 1-3: When a new value is assigned to a variable, the old one is forgotten.

Variable Names

Table 1-3 has examples of legal variable names. You can name a variable anything as long as it obeys the following three rules:

1. It can be only one word.
2. It can use only letters, numbers, and the underscore (_) character.
3. It can't begin with a number.

Table 1-3: Valid and Invalid Variable Names

Valid variable names	Invalid variable names
`balance`	`current-balance` (hyphens are not allowed)
`currentBalance`	`current balance` (spaces are not allowed)
`current_balance`	`4account` (can't begin with a number)
`_spam`	`42` (can't begin with a number)
`SPAM`	`total_$um` (special characters like `$` are not allowed)
`account4`	`'hello'` (special characters like `'` are not allowed)

Variable names are case-sensitive, meaning that spam, SPAM, Spam, and sPaM are four different variables. It is a Python convention to start your variables with a lowercase letter.

This book uses camelcase for variable names instead of underscores; that is, variables lookLikeThis instead of looking_like_this. Some experienced programmers may point out that the official Python code style, PEP 8, says that underscores should be used. I unapologetically prefer camelcase and point to "A Foolish Consistency Is the Hobgoblin of Little Minds" in PEP 8 itself:

> "Consistency with the style guide is important. But most importantly: know when to be inconsistent—sometimes the style guide just doesn't apply. When in doubt, use your best judgment."

A good variable name describes the data it contains. Imagine that you moved to a new house and labeled all of your moving boxes as *Stuff*. You'd never find anything! The variable names spam, eggs, and bacon are used as generic names for the examples in this book and in much of Python's documentation (inspired by the Monty Python "Spam" sketch), but in your programs, a descriptive name will help make your code more readable.

Your First Program

While the interactive shell is good for running Python instructions one at a time, to write entire Python programs, you'll type the instructions into the file editor. The *file editor* is similar to text editors such as Notepad or TextMate, but it has some specific features for typing in source code. To open the file editor in IDLE, select **File ▸ New Window**.

The window that appears should contain a cursor awaiting your input, but it's different from the interactive shell, which runs Python instructions as soon as you press ENTER. The file editor lets you type in many instructions, save the file, and run the program. Here's how you can tell the difference between the two:

- The interactive shell window will always be the one with the >>> prompt.
- The file editor window will not have the >>> prompt.

Now it's time to create your first program! When the file editor window opens, type the following into it:

```
❶ # This program says hello and asks for my name.

❷ print('Hello world!')
  print('What is your name?')    # ask for their name
❸ myName = input()
❹ print('It is good to meet you, ' + myName)
❺ print('The length of your name is:')
  print(len(myName))
```

```
❻ print('What is your age?')    # ask for their age
myAge = input()
print('You will be ' + str(int(myAge) + 1) + ' in a year.')
```

Once you've entered your source code, save it so that you won't have to retype it each time you start IDLE. From the menu at the top of the file editor window, select **File ▸ Save As**. In the Save As window, enter **`hello.py`** in the File Name field and then click **Save**.

You should save your programs every once in a while as you type them. That way, if the computer crashes or you accidentally exit from IDLE, you won't lose the code. As a shortcut, you can press CTRL-S on Windows and Linux or ⌘-S on OS X to save your file.

Once you've saved, let's run our program. Select **Run ▸ Run Module** or just press the **F5** key. Your program should run in the interactive shell window that appeared when you first started IDLE. Remember, you have to press F5 from the file editor window, not the interactive shell window. Enter your name when your program asks for it. The program's output in the interactive shell should look something like this:

```
Python 3.3.2 (v3.3.2:d047928ae3f6, May 16 2013, 00:06:53) [MSC v.1600 64 bit
(AMD64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> ================================ RESTART ================================
>>>
Hello world!
What is your name?
Al
It is good to meet you, Al
The length of your name is:
2
What is your age?
4
You will be 5 in a year.
>>>
```

When there are no more lines of code to execute, the Python program *terminates*; that is, it stops running. (You can also say that the Python program *exits*.)

You can close the file editor by clicking the X at the top of the window. To reload a saved program, select **File ▸ Open** from the menu. Do that now, and in the window that appears, choose ***hello.py***, and click the **Open** button. Your previously saved *hello.py* program should open in the file editor window.

Dissecting Your Program

With your new program open in the file editor, let's take a quick tour of the Python instructions it uses by looking at what each line of code does.

Comments

The following line is called a *comment*.

```
❶ # This program says hello and asks for my name.
```

Python ignores comments, and you can use them to write notes or remind yourself what the code is trying to do. Any text for the rest of the line following a hash mark (#) is part of a comment.

Sometimes, programmers will put a # in front of a line of code to temporarily remove it while testing a program. This is called *commenting out* code, and it can be useful when you're trying to figure out why a program doesn't work. You can remove the # later when you are ready to put the line back in.

Python also ignores the blank line after the comment. You can add as many blank lines to your program as you want. This can make your code easier to read, like paragraphs in a book.

The print() Function

The `print()` function displays the string value inside the parentheses on the screen.

```
❷ print('Hello world!')
  print('What is your name?') # ask for their name
```

The line `print('Hello world!')` means "Print out the text in the string `'Hello world!'`." When Python executes this line, you say that Python is *calling* the `print()` function and the string value is being *passed* to the function. A value that is passed to a function call is an *argument*. Notice that the quotes are not printed to the screen. They just mark where the string begins and ends; they are not part of the string value.

NOTE *You can also use this function to put a blank line on the screen; just call* `print()` *with nothing in between the parentheses.*

When writing a function name, the opening and closing parentheses at the end identify it as the name of a function. This is why in this book you'll see `print()` rather than `print`. Chapter 2 describes functions in more detail.

The input() Function

The `input()` function waits for the user to type some text on the keyboard and press ENTER.

```
❸ myName = input()
```

This function call evaluates to a string equal to the user's text, and the previous line of code assigns the `myName` variable to this string value.

You can think of the input() function call as an expression that evaluates to whatever string the user typed in. If the user entered 'Al', then the expression would evaluate to myName = 'Al'.

Printing the User's Name

The following call to print() actually contains the expression 'It is good to meet you, ' + myName between the parentheses.

```
❹ print('It is good to meet you, ' + myName)
```

Remember that expressions can always evaluate to a single value. If 'Al' is the value stored in myName on the previous line, then this expression evaluates to 'It is good to meet you, Al'. This single string value is then passed to print(), which prints it on the screen.

The len() Function

You can pass the len() function a string value (or a variable containing a string), and the function evaluates to the integer value of the number of characters in that string.

```
❺ print('The length of your name is:')
print(len(myName))
```

Enter the following into the interactive shell to try this:

```
>>> len('hello')
5
>>> len('My very energetic monster just scarfed nachos.')
46
>>> len('')
0
```

Just like those examples, len(myName) evaluates to an integer. It is then passed to print() to be displayed on the screen. Notice that print() allows you to pass it either integer values or string values. But notice the error that shows up when you type the following into the interactive shell:

```
>>> print('I am ' + 29 + ' years old.')
Traceback (most recent call last):
  File "<pyshell#6>", line 1, in <module>
    print('I am ' + 29 + ' years old.')
TypeError: Can't convert 'int' object to str implicitly
```

The print() function isn't causing that error, but rather it's the expression you tried to pass to print(). You get the same error message if you type the expression into the interactive shell on its own.

```
>>> 'I am ' + 29 + ' years old.'
Traceback (most recent call last):
  File "<pyshell#7>", line 1, in <module>
    'I am ' + 29 + ' years old.'
TypeError: Can't convert 'int' object to str implicitly
```

Python gives an error because you can use the + operator only to add two integers together or concatenate two strings. You can't add an integer to a string because this is ungrammatical in Python. You can fix this by using a string version of the integer instead, as explained in the next section.

The str(), int(), and float() Functions

If you want to concatenate an integer such as `29` with a string to pass to `print()`, you'll need to get the value `'29'`, which is the string form of `29`. The `str()` function can be passed an integer value and will evaluate to a string value version of it, as follows:

```
>>> str(29)
'29'
>>> print('I am ' + str(29) + ' years old.')
I am 29 years old.
```

Because `str(29)` evaluates to `'29'`, the expression `'I am ' + str(29) + ' years old.'` evaluates to `'I am ' + '29' + ' years old.'`, which in turn evaluates to `'I am 29 years old.'`. This is the value that is passed to the `print()` function.

The `str()`, `int()`, and `float()` functions will evaluate to the string, integer, and floating-point forms of the value you pass, respectively. Try converting some values in the interactive shell with these functions, and watch what happens.

```
>>> str(0)
'0'
>>> str(-3.14)
'-3.14'
>>> int('42')
42
>>> int('-99')
-99
>>> int(1.25)
1
>>> int(1.99)
1
>>> float('3.14')
3.14
>>> float(10)
10.0
```

The previous examples call the `str()`, `int()`, and `float()` functions and pass them values of the other data types to obtain a string, integer, or floating-point form of those values.

The `str()` function is handy when you have an integer or float that you want to concatenate to a string. The `int()` function is also helpful if you have a number as a string value that you want to use in some mathematics. For example, the `input()` function always returns a string, even if the user enters a number. Enter **`spam = input()`** into the interactive shell and enter **`101`** when it waits for your text.

```
>>> spam = input()
101
>>> spam
'101'
```

The value stored inside `spam` isn't the integer `101` but the string `'101'`. If you want to do math using the value in `spam`, use the `int()` function to get the integer form of `spam` and then store this as the new value in `spam`.

```
>>> spam = int(spam)
>>> spam
101
```

Now you should be able to treat the `spam` variable as an integer instead of a string.

```
>>> spam * 10 / 5
202.0
```

Note that if you pass a value to `int()` that it cannot evaluate as an integer, Python will display an error message.

```
>>> int('99.99')
Traceback (most recent call last):
  File "<pyshell#18>", line 1, in <module>
    int('99.99')
ValueError: invalid literal for int() with base 10: '99.99'
>>> int('twelve')
Traceback (most recent call last):
  File "<pyshell#19>", line 1, in <module>
    int('twelve')
ValueError: invalid literal for int() with base 10: 'twelve'
```

The `int()` function is also useful if you need to round a floating-point number down.

```
>>> int(7.7)
7
>>> int(7.7) + 1
8
```

In your program, you used the `int()` and `str()` functions in the last three lines to get a value of the appropriate data type for the code.

```
❻ print('What is your age?') # ask for their age
myAge = input()
print('You will be ' + str(int(myAge) + 1) + ' in a year.')
```

The `myAge` variable contains the value returned from `input()`. Because the `input()` function always returns a string (even if the user typed in a number), you can use the `int(myAge)` code to return an integer value of the string in `myAge`. This integer value is then added to `1` in the expression `int(myAge) + 1`.

The result of this addition is passed to the `str()` function: `str(int(myAge) + 1)`. The string value returned is then concatenated with the strings `'You will be '` and `' in a year.'` to evaluate to one large string value. This large string is finally passed to `print()` to be displayed on the screen.

Let's say the user enters the string `'4'` for `myAge`. The string `'4'` is converted to an integer, so you can add one to it. The result is `5`. The `str()` function converts the result back to a string, so you can concatenate it with the second string, `'in a year.'`, to create the final message. These evaluation steps would look something like Figure 1-4.

TEXT AND NUMBER EQUIVALENCE

Although the string value of a number is considered a completely different value from the integer or floating-point version, an integer can be equal to a floating point.

```
>>> 42 == '42'
False
>>> 42 == 42.0
True
>>> 42.0 == 0042.000
True
```

Python makes this distinction because strings are text, while integers and floats are both numbers.

```
print('You will be ' + str(int(myAge) + 1) + ' in a year.')

print('You will be ' + str(int( '4' ) + 1) + ' in a year.')

print('You will be ' + str(    4 + 1     ) + ' in a year.')

print('You will be ' + str(      5       ) + ' in a year.')

print('You will be ' +         '5'         + ' in a year.')

print('You will be 5'                      + ' in a year.')

print('You will be 5 in a year.')
```

Figure 1-4: The evaluation steps, if `4` was stored in `myAge`

Summary

You can compute expressions with a calculator or type string concatenations with a word processor. You can even do string replication easily by copying and pasting text. But expressions, and their component values—operators, variables, and function calls—are the basic building blocks that make programs. Once you know how to handle these elements, you will be able to instruct Python to operate on large amounts of data for you.

It is good to remember the different types of operators (`+`, `-`, `*`, `/`, `//`, `%`, and `**` for math operations, and `+` and `*` for string operations) and the three data types (integers, floating-point numbers, and strings) introduced in this chapter.

A few different functions were introduced as well. The `print()` and `input()` functions handle simple text output (to the screen) and input (from the keyboard). The `len()` function takes a string and evaluates to an int of the number of characters in the string. The `str()`, `int()`, and `float()` functions will evaluate to the string, integer, or floating-point number form of the value they are passed.

In the next chapter, you will learn how to tell Python to make intelligent decisions about what code to run, what code to skip, and what code to repeat based on the values it has. This is known as *flow control*, and it allows you to write programs that make intelligent decisions.

Practice Questions

1. Which of the following are operators, and which are values?

```
*
'hello'
-88.8
-
/
+
5
```

2. Which of the following is a variable, and which is a string?

```
spam
'spam'
```

3. Name three data types.
4. What is an expression made up of? What do all expressions do?
5. This chapter introduced assignment statements, like `spam = 10`. What is the difference between an expression and a statement?
6. What does the variable `bacon` contain after the following code runs?

```
bacon = 20
bacon + 1
```

7. What should the following two expressions evaluate to?

```
'spam' + 'spamspam'
'spam' * 3
```

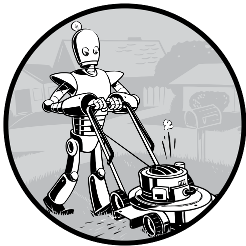
8. Why is `eggs` a valid variable name while `100` is invalid?
9. What three functions can be used to get the integer, floating-point number, or string version of a value?
10. Why does this expression cause an error? How can you fix it?

```
'I have eaten ' + 99 + ' burritos.'
```

Extra credit: Search online for the Python documentation for the `len()` function. It will be on a web page titled "Built-in Functions." Skim the list of other functions Python has, look up what the `round()` function does, and experiment with it in the interactive shell.

2

FLOW CONTROL



So you know the basics of individual instructions and that a program is just a series of instructions. But the real strength of programming isn't just running (or *executing*) one instruction after another like a weekend errand list. Based on how the expressions evaluate, the program can decide to skip instructions, repeat them, or choose one of several instructions to run. In fact, you almost never want your programs to start from the first line of code and simply execute every line, straight to the end. *Flow control statements* can decide which Python instructions to execute under which conditions.

These flow control statements directly correspond to the symbols in a flowchart, so I'll provide flowchart versions of the code discussed in this chapter. Figure 2-1 shows a flowchart for what to do if it's raining. Follow the path made by the arrows from Start to End.

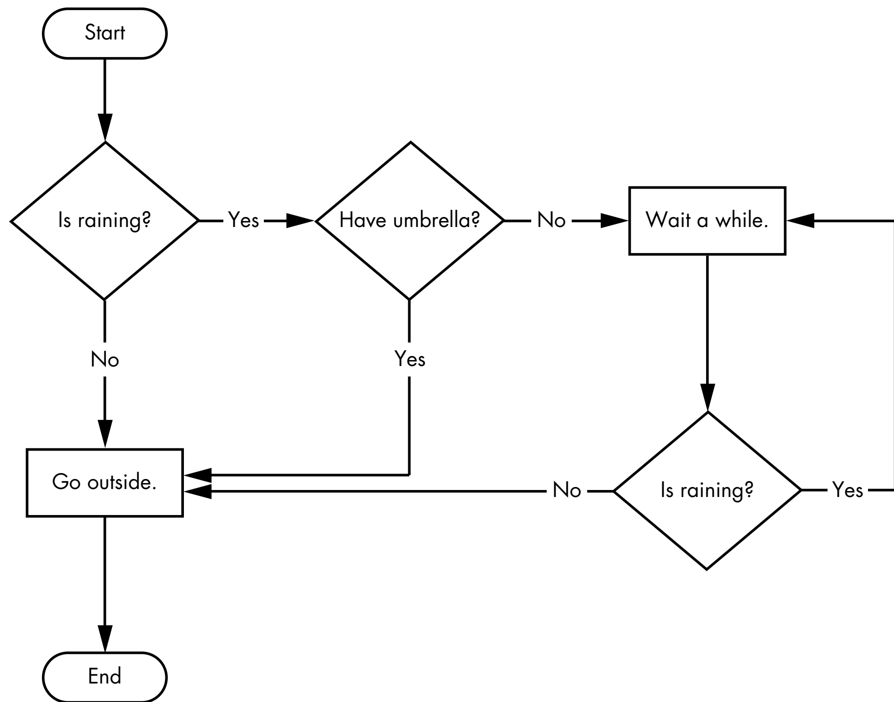


Figure 2-1: A flowchart to tell you what to do if it is raining

In a flowchart, there is usually more than one way to go from the start to the end. The same is true for lines of code in a computer program. Flowcharts represent these branching points with diamonds, while the other steps are represented with rectangles. The starting and ending steps are represented with rounded rectangles.

But before you learn about flow control statements, you first need to learn how to represent those *yes* and *no* options, and you need to understand how to write those branching points as Python code. To that end, let's explore Boolean values, comparison operators, and Boolean operators.

Boolean Values

While the integer, floating-point, and string data types have an unlimited number of possible values, the *Boolean* data type has only two values: `True` and `False`. (Boolean is capitalized because the data type is named after mathematician George Boole.) When typed as Python code, the Boolean values `True` and `False` lack the quotes you place around strings, and they always start with a capital *T* or *F*, with the rest of the word in lowercase. Enter the following into the interactive shell. (Some of these instructions are intentionally incorrect, and they'll cause error messages to appear.)

```
❶ >>> spam = True
   >>> spam
   True
❷ >>> true
   Traceback (most recent call last):
     File "<pyshell#2>", line 1, in <module>
       true
   NameError: name 'true' is not defined
❸ >>> True = 2 + 2
   SyntaxError: assignment to keyword
```

Like any other value, Boolean values are used in expressions and can be stored in variables ❶. If you don't use the proper case ❷ or you try to use `True` and `False` for variable names ❸, Python will give you an error message.

Comparison Operators

Comparison operators compare two values and evaluate down to a single Boolean value. Table 2-1 lists the comparison operators.

Table 2-1: Comparison Operators

Operator	Meaning
`==`	Equal to
`!=`	Not equal to
`<`	Less than
`>`	Greater than
`<=`	Less than or equal to
`>=`	Greater than or equal to

These operators evaluate to `True` or `False` depending on the values you give them. Let's try some operators now, starting with `==` and `!=`.

```
>>> 42 == 42
True
>>> 42 == 99
False
>>> 2 != 3
True
>>> 2 != 2
False
```

As you might expect, `==` (equal to) evaluates to `True` when the values on both sides are the same, and `!=` (not equal to) evaluates to `True` when the two values are different. The `==` and `!=` operators can actually work with values of any data type.

```
>>> 'hello' == 'hello'
True
>>> 'hello' == 'Hello'
False
>>> 'dog' != 'cat'
True
>>> True == True
True
>>> True != False
True
>>> 42 == 42.0
True
❶ >>> 42 == '42'
False
```

Note that an integer or floating-point value will always be unequal to a string value. The expression `42 == '42'` ❶ evaluates to `False` because Python considers the integer `42` to be different from the string `'42'`.

The `<`, `>`, `<=`, and `>=` operators, on the other hand, work properly only with integer and floating-point values.

```
>>> 42 < 100
True
>>> 42 > 100
False
>>> 42 < 42
False
>>> eggCount = 42
❶ >>> eggCount <= 42
True
>>> myAge = 29
❷ >>> myAge >= 10
True
```

THE DIFFERENCE BETWEEN THE == AND = OPERATORS

You might have noticed that the == operator (equal to) has two equal signs, while the = operator (assignment) has just one equal sign. It's easy to confuse these two operators with each other. Just remember these points:

- The == operator (equal to) asks whether two values are the same as each other.
- The = operator (assignment) puts the value on the right into the variable on the left.

To help remember which is which, notice that the == operator (equal to) consists of two characters, just like the != operator (not equal to) consists of two characters.

You'll often use comparison operators to compare a variable's value to some other value, like in the `eggCount <= 42` ❶ and `myAge >= 10` ❷ examples. (After all, instead of typing `'dog' != 'cat'` in your code, you could have just typed `True`.) You'll see more examples of this later when you learn about flow control statements.

Boolean Operators

The three Boolean operators (`and`, `or`, and `not`) are used to compare Boolean values. Like comparison operators, they evaluate these expressions down to a Boolean value. Let's explore these operators in detail, starting with the `and` operator.

Binary Boolean Operators

The `and` and `or` operators always take two Boolean values (or expressions), so they're considered *binary* operators. The `and` operator evaluates an expression to `True` if *both* Boolean values are `True`; otherwise, it evaluates to `False`. Enter some expressions using `and` into the interactive shell to see it in action.

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

A *truth table* shows every possible result of a Boolean operator. Table 2-2 is the truth table for the `and` operator.

Table 2-2: The `and` Operator's Truth Table

Expression	Evaluates to...
`True and True`	`True`
`True and False`	`False`
`False and True`	`False`
`False and False`	`False`

On the other hand, the `or` operator evaluates an expression to `True` if *either* of the two Boolean values is `True`. If both are `False`, it evaluates to `False`.

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

You can see every possible outcome of the `or` operator in its truth table, shown in Table 2-3.

Table 2-3: The `or` Operator's Truth Table

Expression	Evaluates to...
`True or True`	`True`
`True or False`	`True`
`False or True`	`True`
`False or False`	`False`

The not Operator

Unlike `and` and `or`, the `not` operator operates on only one Boolean value (or expression). The `not` operator simply evaluates to the opposite Boolean value.

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

Much like using double negatives in speech and writing, you can nest `not` operators ❶, though there's never not no reason to do this in real programs. Table 2-4 shows the truth table for `not`.

Table 2-4: The `not` Operator's Truth Table

Expression	Evaluates to...
`not True`	`False`
`not False`	`True`

Mixing Boolean and Comparison Operators

Since the comparison operators evaluate to Boolean values, you can use them in expressions with the Boolean operators.

Recall that the `and`, `or`, and `not` operators are called Boolean operators because they always operate on the Boolean values `True` and `False`. While expressions like `4 < 5` aren't Boolean values, they are expressions that evaluate down to Boolean values. Try entering some Boolean expressions that use comparison operators into the interactive shell.

```
>>> (4 < 5) and (5 < 6)
True
>>> (4 < 5) and (9 < 6)
False
>>> (1 == 2) or (2 == 2)
True
```

The computer will evaluate the left expression first, and then it will evaluate the right expression. When it knows the Boolean value for each, it will then evaluate the whole expression down to one Boolean value. You can think of the computer's evaluation process for `(4 < 5) and (5 < 6)` as shown in Figure 2-2.

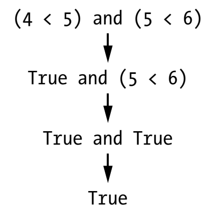


Figure 2-2: The process of evaluating `(4 < 5) and (5 < 6)` to `True`.

You can also use multiple Boolean operators in an expression, along with the comparison operators.

```
>>> 2 + 2 == 4 and not 2 + 2 == 5 and 2 * 2 == 2 + 2
True
```

The Boolean operators have an order of operations just like the math operators do. After any math and comparison operators evaluate, Python evaluates the `not` operators first, then the `and` operators, and then the `or` operators.

Elements of Flow Control

Flow control statements often start with a part called the *condition*, and all are followed by a block of code called the *clause*. Before you learn about Python's specific flow control statements, I'll cover what a condition and a block are.

Conditions

The Boolean expressions you've seen so far could all be considered conditions, which are the same thing as expressions; *condition* is just a more specific name in the context of flow control statements. Conditions always evaluate down to a Boolean value, `True` or `False`. A flow control statement decides what to do based on whether its condition is `True` or `False`, and almost every flow control statement uses a condition.

Blocks of Code

Lines of Python code can be grouped together in *blocks*. You can tell when a block begins and ends from the indentation of the lines of code. There are three rules for blocks.

1. Blocks begin when the indentation increases.
2. Blocks can contain other blocks.
3. Blocks end when the indentation decreases to zero or to a containing block's indentation.

Blocks are easier to understand by looking at some indented code, so let's find the blocks in part of a small game program, shown here:

```
  if name == 'Mary':
❶     print('Hello Mary')
  if password == 'swordfish':
❷     print('Access granted.')
  else:
❸     print('Wrong password.')
```

The first block of code ❶ starts at the line `print('Hello Mary')` and contains all the lines after it. Inside this block is another block ❷, which has only a single line in it: `print('Access Granted.')`. The third block ❸ is also one line long: `print('Wrong password.')`.

Program Execution

In the previous chapter's *hello.py* program, Python started executing instructions at the top of the program going down, one after another. The *program execution* (or simply, *execution*) is a term for the current instruction being executed. If you print the source code on paper and put your finger on each line as it is executed, you can think of your finger as the program execution.

Not all programs execute by simply going straight down, however. If you use your finger to trace through a program with flow control statements, you'll likely find yourself jumping around the source code based on conditions, and you'll probably skip entire clauses.

Flow Control Statements

Now, let's explore the most important piece of flow control: the statements themselves. The statements represent the diamonds you saw in the flowchart in Figure 2-1, and they are the actual decisions your programs will make.

if Statements

The most common type of flow control statement is the `if` statement. An `if` statement's clause (that is, the block following the `if` statement) will execute if the statement's condition is `True`. The clause is skipped if the condition is `False`.

In plain English, an `if` statement could be read as, "If this condition is true, execute the code in the clause." In Python, an `if` statement consists of the following:

- The `if` keyword
- A condition (that is, an expression that evaluates to `True` or `False`)
- A colon
- Starting on the next line, an indented block of code (called the `if` clause)

For example, let's say you have some code that checks to see whether someone's name is Alice. (Pretend `name` was assigned some value earlier.)

```
if name == 'Alice':
    print('Hi, Alice.')
```

All flow control statements end with a colon and are followed by a new block of code (the clause). This `if` statement's clause is the block with `print('Hi, Alice.')`. Figure 2-3 shows what a flowchart of this code would look like.

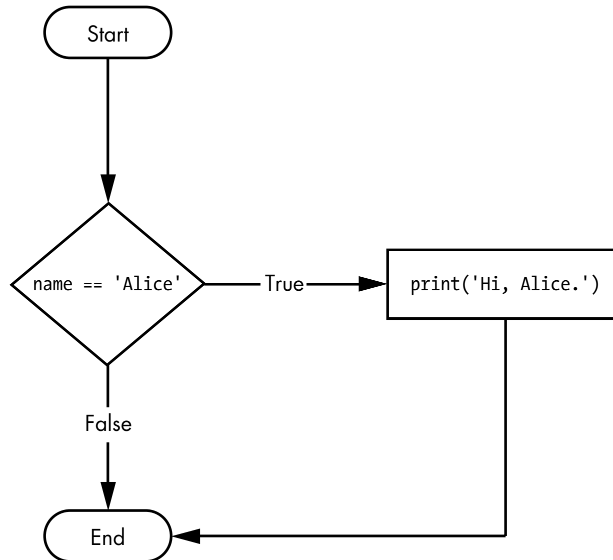


Figure 2-3: The flowchart for an `if` statement

else Statements

An `if` clause can optionally be followed by an `else` statement. The `else` clause is executed only when the `if` statement's condition is `False`. In plain English, an `else` statement could be read as, "If this condition is true, execute this code. Or else, execute that code." An `else` statement doesn't have a condition, and in code, an `else` statement always consists of the following:

- The `else` keyword
- A colon
- Starting on the next line, an indented block of code (called the `else` clause)

Returning to the Alice example, let's look at some code that uses an `else` statement to offer a different greeting if the person's name isn't Alice.

```
if name == 'Alice':
    print('Hi, Alice.')
```

```
else:
    print('Hello, stranger.')
```

Figure 2-4 shows what a flowchart of this code would look like.

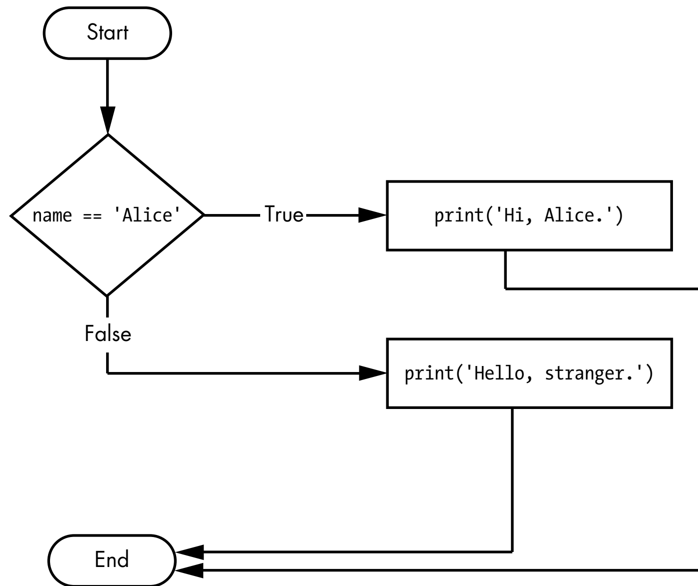


Figure 2-4: The flowchart for an `else` statement

elif Statements

While only one of the `if` or `else` clauses will execute, you may have a case where you want one of *many* possible clauses to execute. The `elif` statement is an "else if" statement that always follows an `if` or another `elif` statement. It provides another condition that is checked only if any of the previous conditions were `False`. In code, an `elif` statement always consists of the following:

- The `elif` keyword
- A condition (that is, an expression that evaluates to `True` or `False`)
- A colon
- Starting on the next line, an indented block of code (called the `elif` clause)

Let's add an `elif` to the name checker to see this statement in action.

```
if name == 'Alice':
    print('Hi, Alice.')
elif age < 12:
    print('You are not Alice, kiddo.')
```

This time, you check the person's age, and the program will tell them something different if they're younger than 12. You can see the flowchart for this in Figure 2-5.

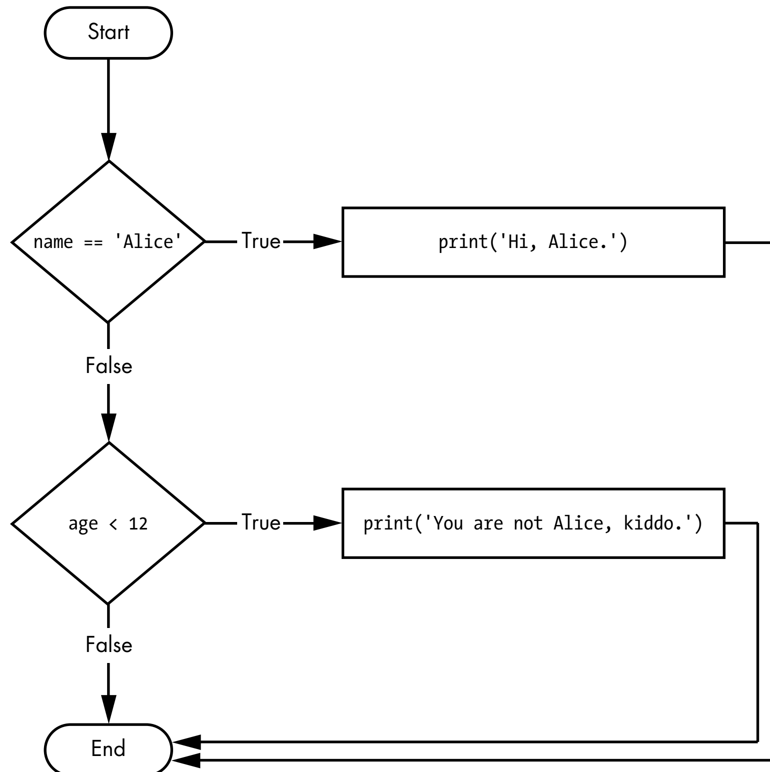


Figure 2-5: The flowchart for an `elif` *statement*

The `elif` clause executes if `age < 12` is `True` and `name == 'Alice'` is `False`. However, if both of the conditions are `False`, then both of the clauses are skipped. It is *not* guaranteed that at least one of the clauses will be executed. When there is a chain of `elif` statements, only one or none of the clauses will be executed. Once one of the statements' conditions is found to be `True`, the rest of the `elif` clauses are automatically skipped. For example, open a new file editor window and enter the following code, saving it as *vampire.py*:

```
if name == 'Alice':
    print('Hi, Alice.')
elif age < 12:
    print('You are not Alice, kiddo.')
elif age > 2000:
    print('Unlike you, Alice is not an undead, immortal vampire.')
elif age > 100:
    print('You are not Alice, grannie.')
```

Here I've added two more `elif` statements to make the name checker greet a person with different answers based on `age`. Figure 2-6 shows the flowchart for this.

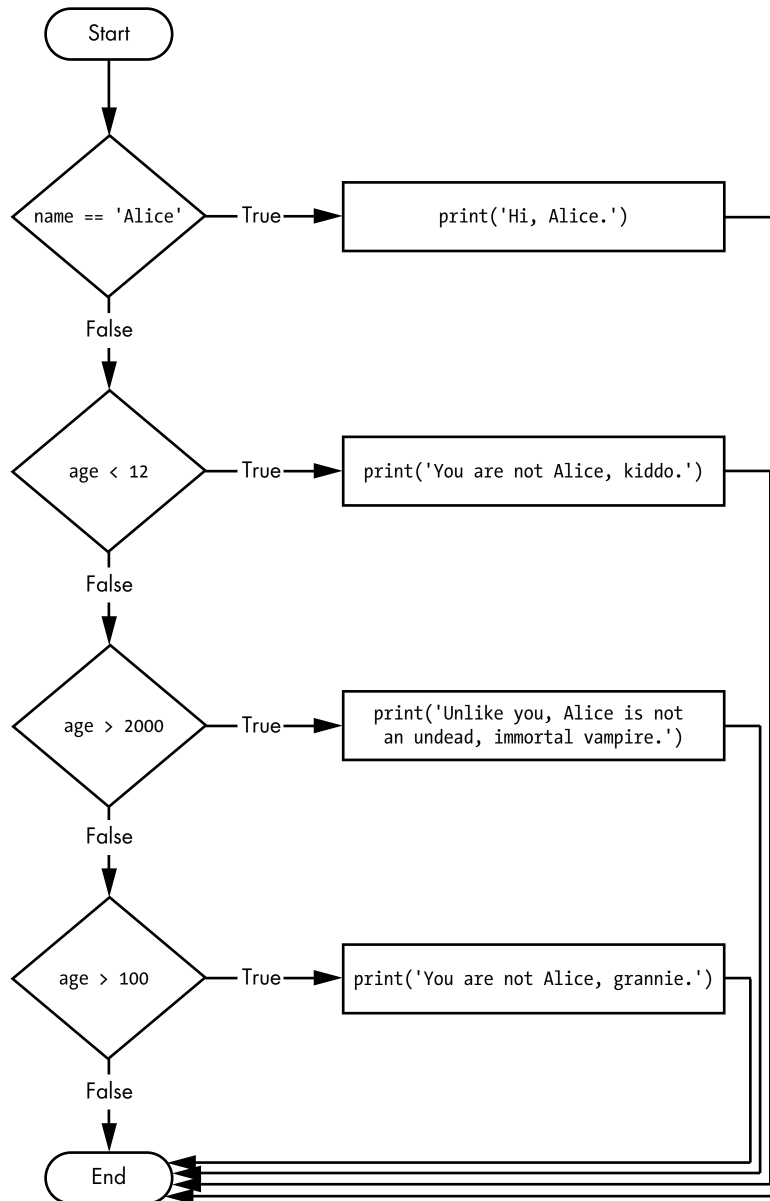


Figure 2-6: The flowchart for multiple `elif` statements in the vampire.py *program*

The order of the `elif` statements does matter, however. Let's rearrange them to introduce a bug. Remember that the rest of the `elif` clauses are automatically skipped once a `True` condition has been found, so if you swap around some of the clauses in *vampire.py*, you run into a problem. Change the code to look like the following, and save it as *vampire2.py*:

```
  if name == 'Alice':
      print('Hi, Alice.')
  elif age < 12:
      print('You are not Alice, kiddo.')
❶ elif age > 100:
      print('You are not Alice, grannie.')
  elif age > 2000:
      print('Unlike you, Alice is not an undead, immortal vampire.')
```

Say the `age` variable contains the value `3000` before this code is executed. You might expect the code to print the string `'Unlike you, Alice is not an undead, immortal vampire.'`. However, because the `age > 100` condition is `True` (after all, 3000 *is* greater than 100) ❶, the string `'You are not Alice, grannie.'` is printed, and the rest of the `elif` statements are automatically skipped. Remember, at most only one of the clauses will be executed, and for `elif` statements, the order matters!

Figure 2-7 shows the flowchart for the previous code. Notice how the diamonds for `age > 100` and `age > 2000` are swapped.

Optionally, you can have an `else` statement after the last `elif` statement. In that case, it *is* guaranteed that at least one (and only one) of the clauses will be executed. If the conditions in every `if` and `elif` statement are `False`, then the `else` clause is executed. For example, let's re-create the Alice program to use `if`, `elif`, and `else` clauses.

```
if name == 'Alice':
    print('Hi, Alice.')
elif age < 12:
    print('You are not Alice, kiddo.')
else:
    print('You are neither Alice nor a little kid.')
```

Figure 2-8 shows the flowchart for this new code, which we'll save as *littleKid.py*.

In plain English, this type of flow control structure would be, "If the first condition is true, do this. Else, if the second condition is true, do that. Otherwise, do something else." When you use all three of these statements together, remember these rules about how to order them to avoid bugs like the one in Figure 2-7. First, there is always exactly one `if` statement. Any `elif` statements you need should follow the `if` statement. Second, if you want to be sure that at least one clause is executed, close the structure with an `else` statement.

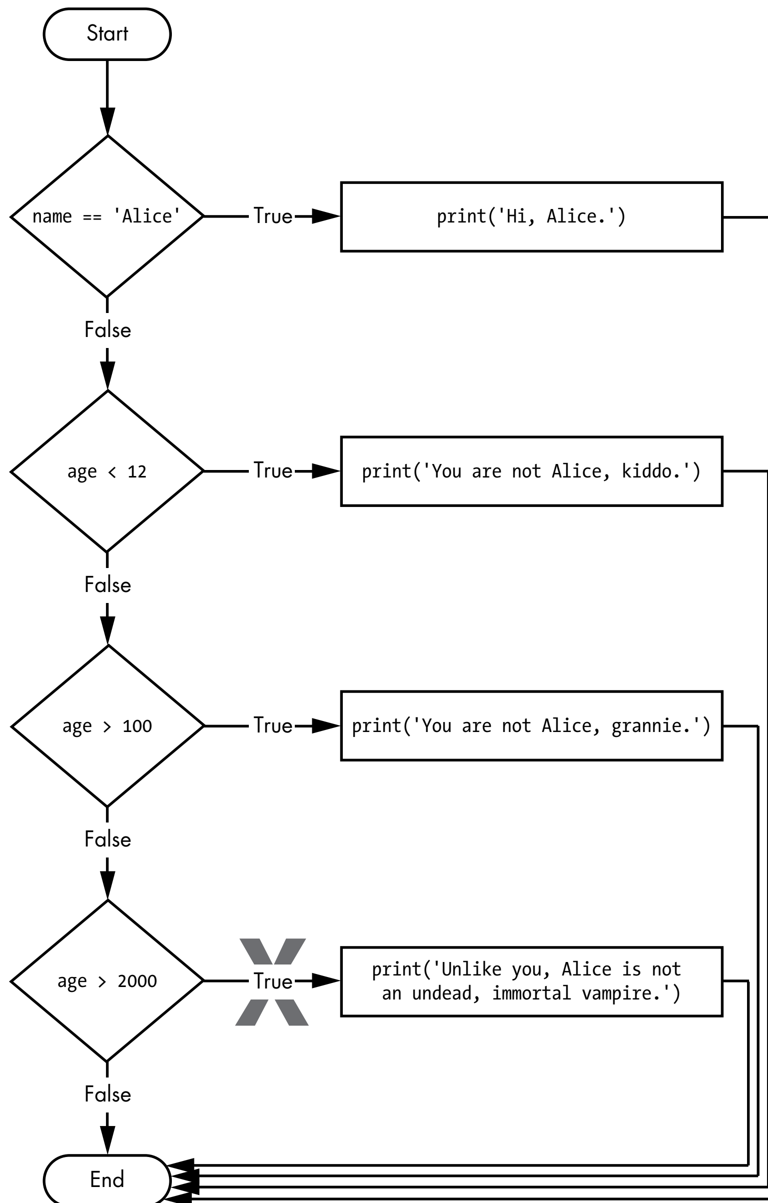


Figure 2-7: The flowchart for the vampire2.py *program. The crossed-out path will logically never happen, because if* `age` *were greater than* `2000`*, it would have already been greater than* `100`*.*

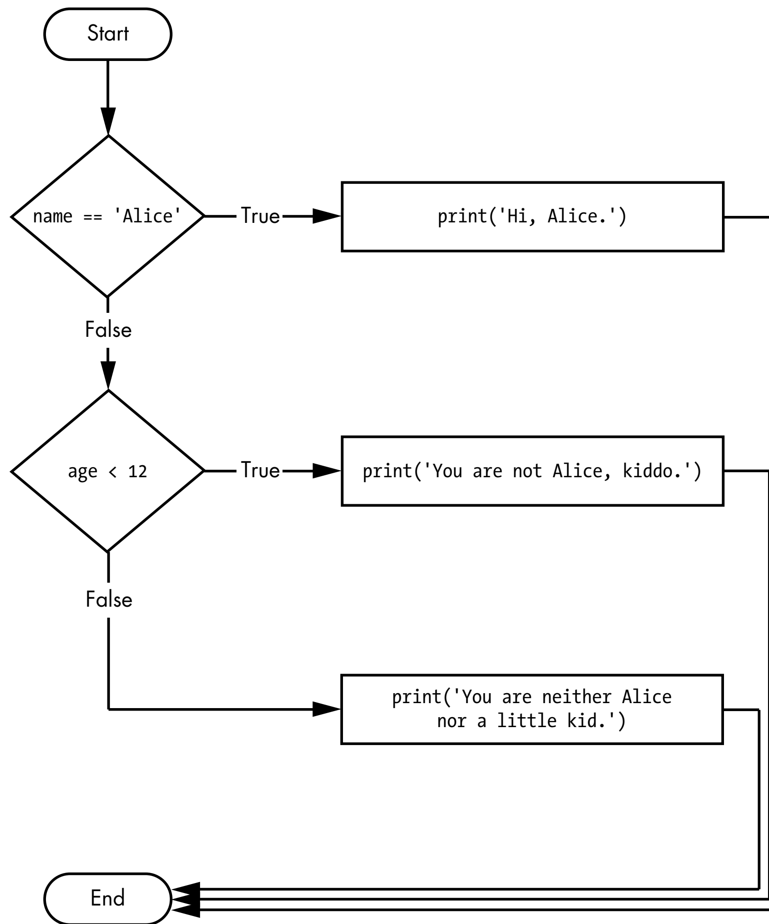


Figure 2-8: Flowchart for the previous littleKid.py *program*

while Loop Statements

You can make a block of code execute over and over again with a `while` statement. The code in a `while` clause will be executed as long as the `while` statement's condition is `True`. In code, a `while` statement always consists of the following:

- The `while` keyword
- A condition (that is, an expression that evaluates to `True` or `False`)
- A colon
- Starting on the next line, an indented block of code (called the `while` clause)

You can see that a `while` statement looks similar to an `if` statement. The difference is in how they behave. At the end of an `if` clause, the program execution continues after the `if` statement. But at the end of a `while` clause, the program execution jumps back to the start of the `while` statement. The `while` clause is often called the *while loop* or just the *loop*.

Let's look at an `if` statement and a `while` loop that use the same condition and take the same actions based on that condition. Here is the code with an `if` statement:

```
spam = 0
if spam < 5:
    print('Hello, world.')
    spam = spam + 1
```

Here is the code with a `while` statement:

```
spam = 0
while spam < 5:
    print('Hello, world.')
    spam = spam + 1
```

These statements are similar—both `if` and `while` check the value of `spam`, and if it's less than five, they print a message. But when you run these two code snippets, something very different happens for each one. For the `if` statement, the output is simply `"Hello, world."`. But for the `while` statement, it's `"Hello, world."` repeated five times! Take a look at the flowcharts for these two pieces of code, Figures 2-9 and 2-10, to see why this happens.

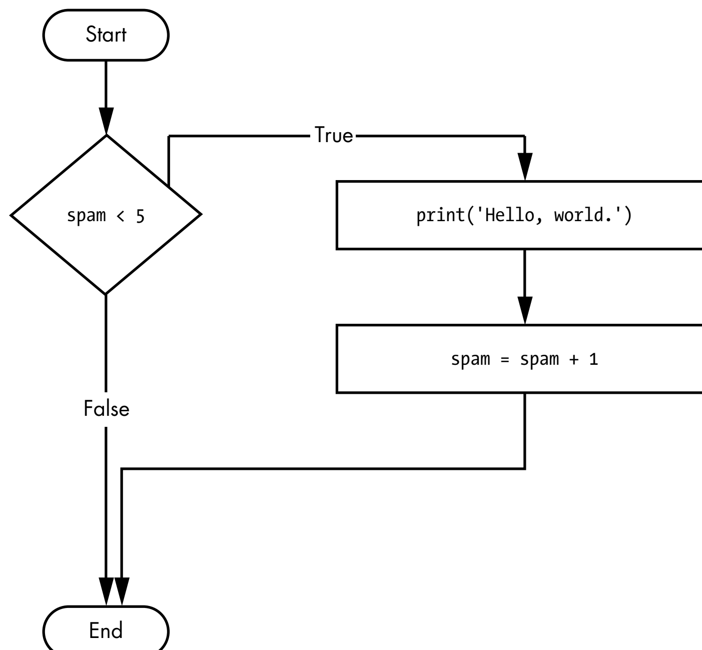


Figure 2-9: The flowchart for the `if` statement code

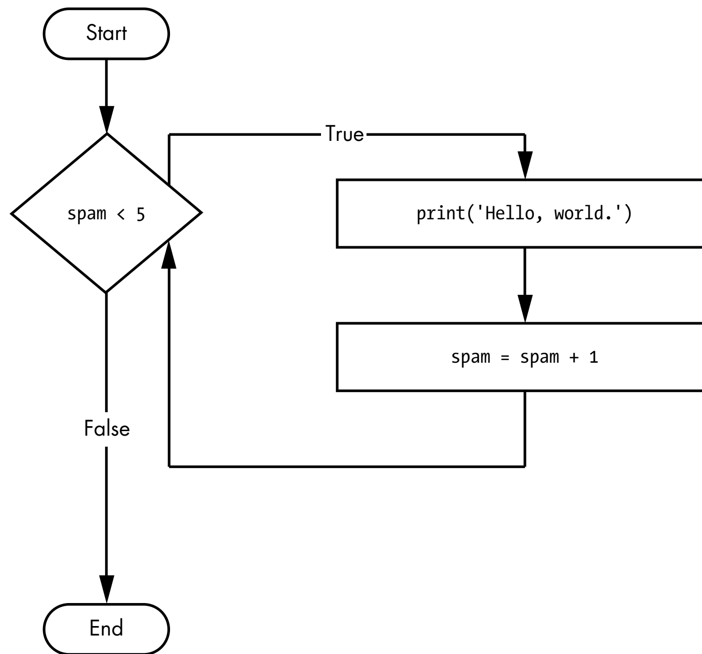


Figure 2-10: The flowchart for the `while` *statement code*

The code with the `if` statement checks the condition, and it prints `Hello, world.` only once if that condition is true. The code with the `while` loop, on the other hand, will print it five times. It stops after five prints because the integer in `spam` is incremented by one at the end of each loop iteration, which means that the loop will execute five times before `spam < 5` is `False`.

In the `while` loop, the condition is always checked at the start of each *iteration* (that is, each time the loop is executed). If the condition is `True`, then the clause is executed, and afterward, the condition is checked again. The first time the condition is found to be `False`, the `while` clause is skipped.

An Annoying while Loop

Here's a small example program that will keep asking you to type, literally, **`your name`**. Select **File ▸ New Window** to open a new file editor window, enter the following code, and save the file as *yourName.py*:

```
❶ name = ''
❷ while name != 'your name':
      print('Please type your name.')
❸     name = input()
❹ print('Thank you!')
```

First, the program sets the `name` variable ❶ to an empty string. This is so that the `name != 'your name'` condition will evaluate to `True` and the program execution will enter the `while` loop's clause ❷.

The code inside this clause asks the user to type their name, which is assigned to the `name` variable ❸. Since this is the last line of the block, the execution moves back to the start of the `while` loop and reevaluates the condition. If the value in `name` is *not equal* to the string `'your name'`, then the condition is `True`, and the execution enters the `while` clause again.

But once the user types **`your name`**, the condition of the `while` loop will be `'your name' != 'your name'`, which evaluates to `False`. The condition is now `False`, and instead of the program execution reentering the `while` loop's clause, it skips past it and continues running the rest of the program ❹. Figure 2-11 shows a flowchart for the *yourName.py* program.

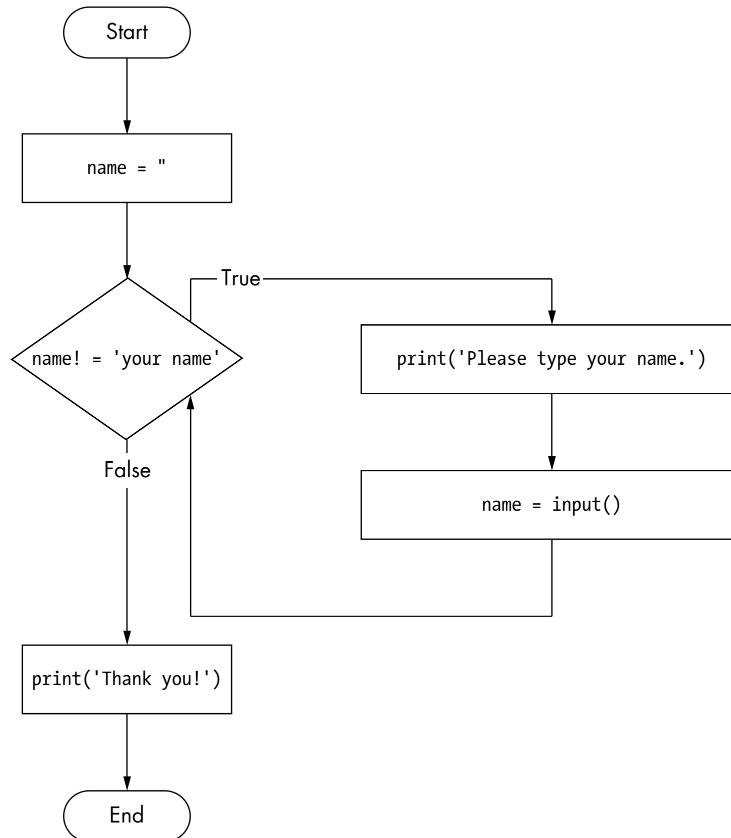


Figure 2-11: A flowchart of the yourName.py *program*

Now, let's see *yourName.py* in action. Press **F5** to run it, and enter something other than **`your name`** a few times before you give the program what it wants.

```
Please type your name.
Al
Please type your name.
Albert
```

```
Please type your name.
%#@#%*(^&!!!
Please type your name.
your name
Thank you!
```

If you never enter **your name**, then the `while` loop's condition will never be `False`, and the program will just keep asking forever. Here, the `input()` call lets the user enter the right string to make the program move on. In other programs, the condition might never actually change, and that can be a problem. Let's look at how you can break out of a `while` loop.

break Statements

There is a shortcut to getting the program execution to break out of a `while` loop's clause early. If the execution reaches a `break` statement, it immediately exits the `while` loop's clause. In code, a `break` statement simply contains the `break` keyword.

Pretty simple, right? Here's a program that does the same thing as the previous program, but it uses a `break` statement to escape the loop. Enter the following code, and save the file as *yourName2.py*:

```
❶ while True:
      print('Please type your name.')
❷     name = input()
❸     if name == 'your name':
❹         break
❺ print('Thank you!')
```

The first line ❶ creates an *infinite loop*; it is a `while` loop whose condition is always `True`. (The expression `True`, after all, always evaluates down to the value `True`.) The program execution will always enter the loop and will exit it only when a `break` statement is executed. (An infinite loop that *never* exits is a common programming bug.)

Just like before, this program asks the user to type **your name** ❷. Now, however, while the execution is still inside the `while` loop, an `if` statement gets executed ❸ to check whether `name` is equal to `your name`. If this condition is `True`, the `break` statement is run ❹, and the execution moves out of the loop to `print('Thank you!')` ❺. Otherwise, the `if` statement's clause with the `break` statement is skipped, which puts the execution at the end of the `while` loop. At this point, the program execution jumps back to the start of the `while` statement ❶ to recheck the condition. Since this condition is merely the `True` Boolean value, the execution enters the loop to ask the user to type **your name** again. See Figure 2-12 for the flowchart of this program.

Run *yourName2.py*, and enter the same text you entered for *yourName.py*. The rewritten program should respond in the same way as the original.

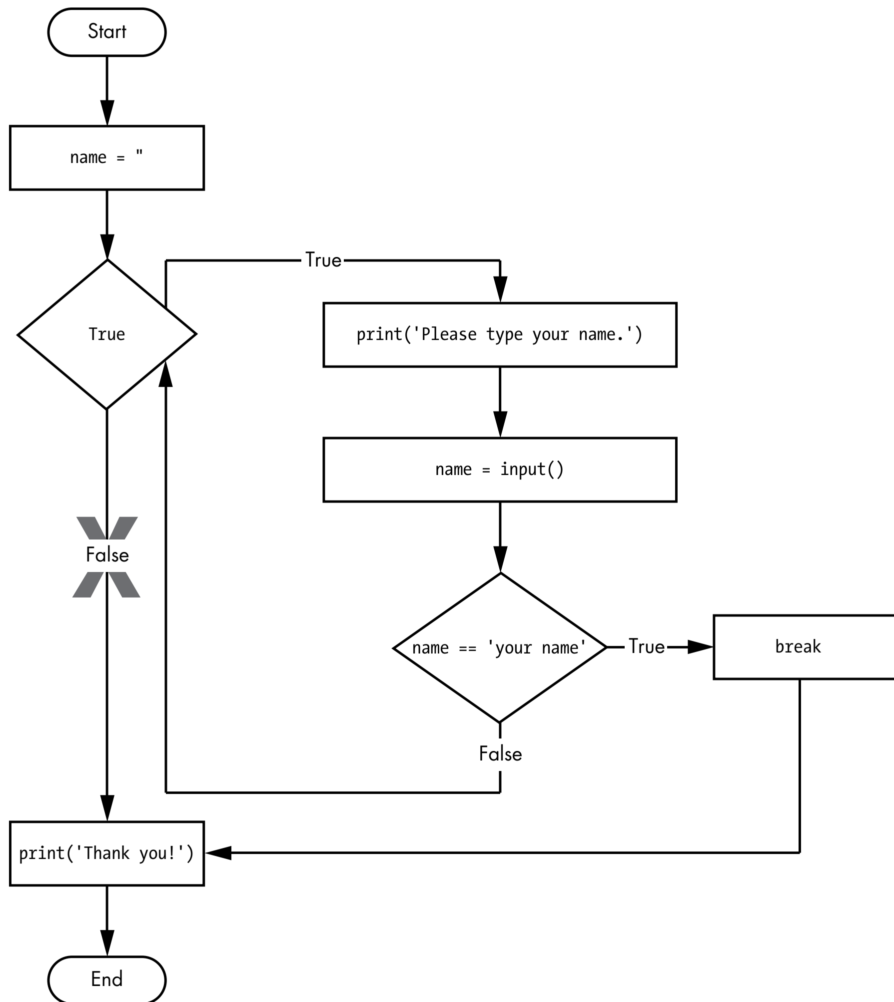


Figure 2-12: The flowchart for the yourName2.py *program with an infinite loop. Note that the X path will logically never happen because the loop condition is always* `True`.

continue Statements

Like `break` statements, `continue` statements are used inside loops. When the program execution reaches a `continue` statement, the program execution immediately jumps back to the start of the loop and reevaluates the loop's condition. (This is also what happens when the execution reaches the end of the loop.)

TRAPPED IN AN INFINITE LOOP?

If you ever run a program that has a bug causing it to get stuck in an infinite loop, press CTRL-C. This will send a `KeyboardInterrupt` error to your program and cause it to stop immediately. To try it, create a simple infinite loop in the file editor, and save it as *infiniteloop.py*.

```
while True:
    print('Hello world!')
```

When you run this program, it will print `Hello world!` to the screen forever, because the `while` statement's condition is always `True`. In IDLE's interactive shell window, there are only two ways to stop this program: press CTRL-C or select **Shell ▸ Restart Shell** from the menu. CTRL-C is handy if you ever want to terminate your program immediately, even if it's not stuck in an infinite loop.

Let's use `continue` to write a program that asks for a name and password. Enter the following code into a new file editor window and save the program as *swordfish.py*.

```
  while True:
      print('Who are you?')
      name = input()
❶     if name != 'Joe':
❷         continue
      print('Hello, Joe. What is the password? (It is a fish.)')
❸     password = input()
      if password == 'swordfish':
❹         break
❺ print('Access granted.')
```

If the user enters any name besides `Joe` ❶, the `continue` statement ❷ causes the program execution to jump back to the start of the loop. When it reevaluates the condition, the execution will always enter the loop, since the condition is simply the value `True`. Once they make it past that `if` statement, the user is asked for a password ❸. If the password entered is `swordfish`, then the `break` statement ❹ is run, and the execution jumps out of the `while` loop to print `Access granted` ❺. Otherwise, the execution continues to the end of the `while` loop, where it then jumps back to the start of the loop. See Figure 2-13 for this program's flowchart.

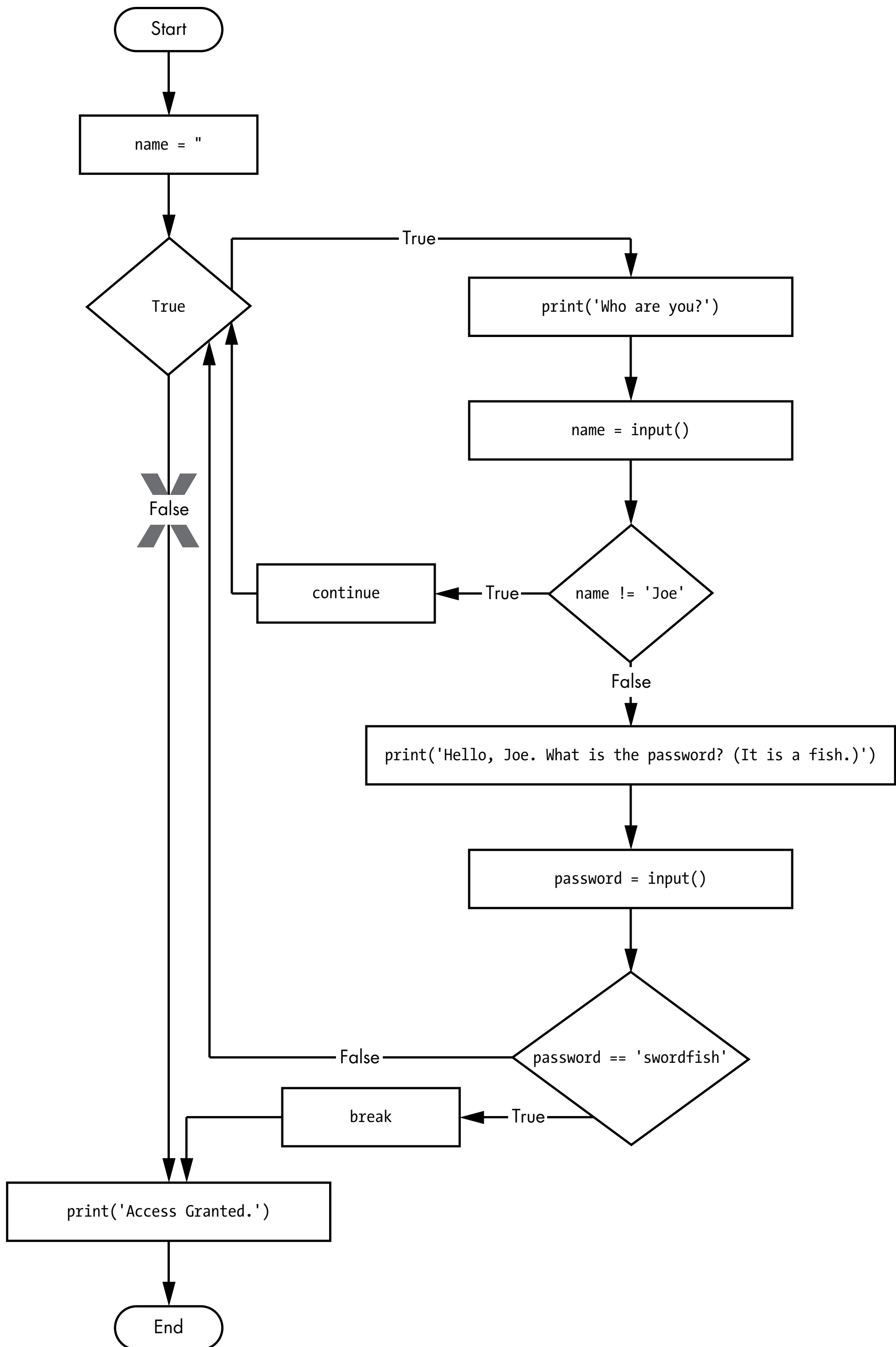


Figure 2-13: A flowchart for swordfish.py*. The X path will logically never happen because the loop condition is always* `True`*.*

"TRUTHY" AND "FALSEY" VALUES

There are some values in other data types that conditions will consider equivalent to `True` and `False`. When used in conditions, `0`, `0.0`, and `''` (the empty string) are considered `False`, while all other values are considered `True`. For example, look at the following program:

```
name = ''
while not name:❶
    print('Enter your name:')
    name = input()
print('How many guests will you have?')
numOfGuests = int(input())
if numOfGuests:❷
    print('Be sure to have enough room for all your guests.')❸
print('Done')
```

If the user enters a blank string for `name`, then the `while` statement's condition will be `True` ❶, and the program continues to ask for a name. If the value for `numOfGuests` is not `0` ❷, then the condition is considered to be `True`, and the program will print a reminder for the user ❸.

You could have typed `not name != ''` instead of `not name`, and `numOfGuests != 0` instead of `numOfGuests`, but using the truthy and falsey values can make your code easier to read.

Run this program and give it some input. Until you claim to be Joe, it shouldn't ask for a password, and once you enter the correct password, it should exit.

```
Who are you?
I'm fine, thanks. Who are you?
Who are you?
Joe
Hello, Joe. What is the password? (It is a fish.)
Mary
Who are you?
Joe
Hello, Joe. What is the password? (It is a fish.)
swordfish
Access granted.
```

for Loops and the range() Function

The `while` loop keeps looping while its condition is `True` (which is the reason for its name), but what if you want to execute a block of code only a certain number of times? You can do this with a `for` loop statement and the `range()` function.

In code, a `for` statement looks something like `for i in range(5):` and always includes the following:

- The `for` keyword
- A variable name
- The `in` keyword
- A call to the `range()` method with up to three integers passed to it
- A colon
- Starting on the next line, an indented block of code (called the `for` clause)

Let's create a new program called *fiveTimes.py* to help you see a `for` loop in action.

```
print('My name is')
for i in range(5):
    print('Jimmy Five Times (' + str(i) + ')')
```

The code in the `for` loop's clause is run five times. The first time it is run, the variable `i` is set to `0`. The `print()` call in the clause will print `Jimmy Five Times (0)`. After Python finishes an iteration through all the code inside the `for` loop's clause, the execution goes back to the top of the loop, and the `for` statement increments `i` by one. This is why `range(5)` results in five iterations through the clause, with `i` being set to `0`, then `1`, then `2`, then `3`, and then `4`. The variable `i` will go up to, but will not include, the integer passed to `range()`. Figure 2-14 shows a flowchart for the *fiveTimes.py* program.

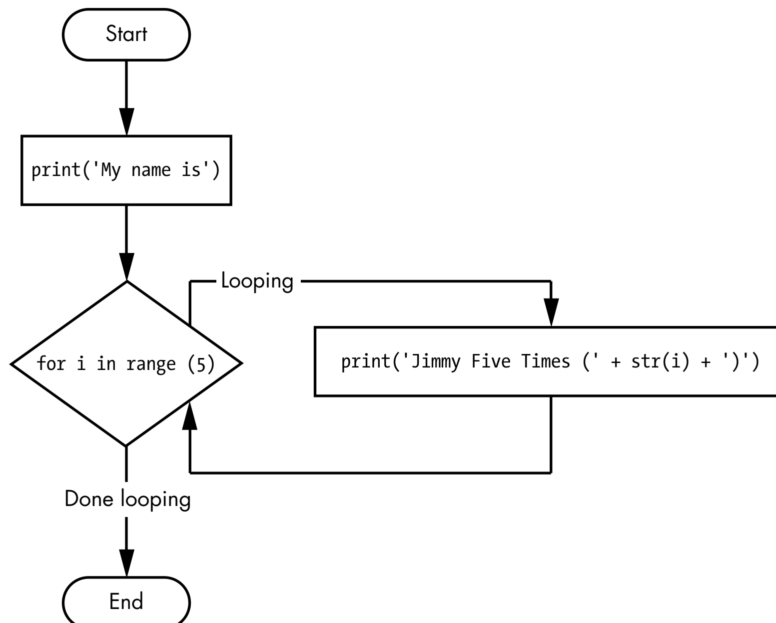


Figure 2-14: The flowchart for fiveTimes.py

When you run this program, it should print `Jimmy Five Times` followed by the value of `i` five times before leaving the `for` loop.

```
My name is
Jimmy Five Times (0)
Jimmy Five Times (1)
Jimmy Five Times (2)
Jimmy Five Times (3)
Jimmy Five Times (4)
```

NOTE *You can use* `break` *and* `continue` *statements inside* `for` *loops as well. The* `continue` *statement will continue to the next value of the* `for` *loop's counter, as if the program execution had reached the end of the loop and returned to the start. In fact, you can use* `continue` *and* `break` *statements only inside* `while` *and* `for` *loops. If you try to use these statements elsewhere, Python will give you an error.*

As another `for` loop example, consider this story about the mathematician Karl Friedrich Gauss. When Gauss was a boy, a teacher wanted to give the class some busywork. The teacher told them to add up all the numbers from 0 to 100. Young Gauss came up with a clever trick to figure out the answer in a few seconds, but you can write a Python program with a `for` loop to do this calculation for you.

```
❶ total = 0
❷ for num in range(101):
❸     total = total + num
❹ print(total)
```

The result should be 5,050. When the program first starts, the `total` variable is set to `0` ❶. The `for` loop ❷ then executes `total = total + num` ❸ 100 times. By the time the loop has finished all of its 100 iterations, every integer from `0` to `100` will have been added to `total`. At this point, `total` is printed to the screen ❹. Even on the slowest computers, this program takes less than a second to complete.

(Young Gauss figured out that there were 50 pairs of numbers that added up to 100: 1 + 99, 2 + 98, 3 + 97, and so on, until 49 + 51. Since 50 × 100 is 5,000, when you add that middle 50, the sum of all the numbers from 0 to 100 is 5,050. Clever kid!)

An Equivalent while Loop

You can actually use a `while` loop to do the same thing as a `for` loop; `for` loops are just more concise. Let's rewrite *fiveTimes.py* to use a `while` loop equivalent of a `for` loop.

```
print('My name is')
i = 0
while i < 5:
    print('Jimmy Five Times (' + str(i) + ')')
    i = i + 1
```

If you run this program, the output should look the same as the *fiveTimes.py* program, which uses a `for` loop.

The Starting, Stopping, and Stepping Arguments to range()

Some functions can be called with multiple arguments separated by a comma, and `range()` is one of them. This lets you change the integer passed to `range()` to follow any sequence of integers, including starting at a number other than zero.

```
for i in range(12, 16):
    print(i)
```

The first argument will be where the `for` loop's variable starts, and the second argument will be up to, but not including, the number to stop at.

```
12
13
14
15
```

The `range()` function can also be called with three arguments. The first two arguments will be the start and stop values, and the third will be the *step argument*. The step is the amount that the variable is increased by after each iteration.

```
for i in range(0, 10, 2):
    print(i)
```

So calling `range(0, 10, 2)` will count from zero to eight by intervals of two.

```
0
2
4
6
8
```

The `range()` function is flexible in the sequence of numbers it produces for `for` loops. *For* example (I never apologize for my puns), you can even use a negative number for the step argument to make the `for` loop count down instead of up.

```
for i in range(5, -1, -1):
    print(i)
```

Running a `for` loop to print `i` with `range(5, -1, -1)` should print from five down to zero.

```
5
4
```

```
3
2
1
0
```

Importing Modules

All Python programs can call a basic set of functions called *built-in functions*, including the `print()`, `input()`, and `len()` functions you've seen before. Python also comes with a set of modules called the *standard library*. Each module is a Python program that contains a related group of functions that can be embedded in your programs. For example, the `math` module has mathematics-related functions, the `random` module has random number–related functions, and so on.

Before you can use the functions in a module, you must import the module with an `import` statement. In code, an `import` statement consists of the following:

- The `import` keyword
- The name of the module
- Optionally, more module names, as long as they are separated by commas

Once you import a module, you can use all the cool functions of that module. Let's give it a try with the `random` module, which will give us access to the `random.ranint()` function.

Enter this code into the file editor, and save it as *printRandom.py*:

```
import random
for i in range(5):
    print(random.randint(1, 10))
```

When you run this program, the output will look something like this:

```
4
1
8
4
1
```

The `random.randint()` function call evaluates to a random integer value between the two integers that you pass it. Since `randint()` is in the `random` module, you must first type **`random.`** in front of the function name to tell Python to look for this function inside the `random` module.

Here's an example of an `import` statement that imports four different modules:

```
import random, sys, os, math
```

Now we can use any of the functions in these four modules. We'll learn more about them later in the book.

from import Statements

An alternative form of the `import` statement is composed of the `from` keyword, followed by the module name, the `import` keyword, and a star; for example, `from random import *`.

With this form of `import` statement, calls to functions in `random` will not need the `random.` prefix. However, using the full name makes for more readable code, so it is better to use the normal form of the `import` statement.

Ending a Program Early with sys.exit()

The last flow control concept to cover is how to terminate the program. This always happens if the program execution reaches the bottom of the instructions. However, you can cause the program to terminate, or exit, by calling the `sys.exit()` function. Since this function is in the `sys` module, you have to import `sys` before your program can use it.

Open a new file editor window and enter the following code, saving it as *exitExample.py*:

```
import sys

while True:
    print('Type exit to exit.')
    response = input()
    if response == 'exit':
        sys.exit()
    print('You typed ' + response + '.')
```

Run this program in IDLE. This program has an infinite loop with no `break` statement inside. The only way this program will end is if the user enters `exit`, causing `sys.exit()` to be called. When `response` is equal to `exit`, the program ends. Since the `response` variable is set by the `input()` function, the user must enter `exit` in order to stop the program.

Summary

By using expressions that evaluate to `True` or `False` (also called *conditions*), you can write programs that make decisions on what code to execute and what code to skip. You can also execute code over and over again in a loop while a certain condition evaluates to `True`. The `break` and `continue` statements are useful if you need to exit a loop or jump back to the start.

These flow control statements will let you write much more intelligent programs. There's another type of flow control that you can achieve by writing your own functions, which is the topic of the next chapter.

Practice Questions

1. What are the two values of the Boolean data type? How do you write them?
2. What are the three Boolean operators?
3. Write out the truth tables of each Boolean operator (that is, every possible combination of Boolean values for the operator and what they evaluate to).
4. What do the following expressions evaluate to?

```
(5 > 4) and (3 == 5)
not (5 > 4)
(5 > 4) or (3 == 5)
not ((5 > 4) or (3 == 5))
(True and True) and (True == False)
(not False) or (not True)
```

5. What are the six comparison operators?
6. What is the difference between the equal to operator and the assignment operator?
7. Explain what a condition is and where you would use one.
8. Identify the three blocks in this code:

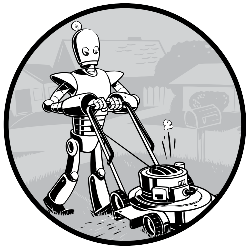
```
spam = 0
if spam == 10:
    print('eggs')
    if spam > 5:
        print('bacon')
    else:
        print('ham')
    print('spam')
print('spam')
```

9. Write code that prints `Hello` if `1` is stored in `spam`, prints `Howdy` if `2` is stored in `spam`, and prints `Greetings!` if anything else is stored in `spam`.
10. What can you press if your program is stuck in an infinite loop?
11. What is the difference between `break` and `continue`?
12. What is the difference between `range(10)`, `range(0, 10)`, and `range(0, 10, 1)` in a `for` loop?
13. Write a short program that prints the numbers `1` to `10` using a `for` loop. Then write an equivalent program that prints the numbers `1` to `10` using a `while` loop.
14. If you had a function named `bacon()` inside a module named `spam`, how would you call it after importing `spam`?

Extra credit: Look up the `round()` and `abs()` functions on the Internet, and find out what they do. Experiment with them in the interactive shell.

3

FUNCTIONS



You're already familiar with the `print()`, `input()`, and `len()` functions from the previous chapters. Python provides several built-in functions like these, but you can also write your own functions. A *function* is like a mini-program within a program.

To better understand how functions work, let's create one. Type this program into the file editor and save it as *helloFunc.py*:

```
❶ def hello():
❷     print('Howdy!')
      print('Howdy!!!')
      print('Hello there.')

❸ hello()
  hello()
  hello()
```

The first line is a `def` statement ❶, which defines a function named `hello()`. The code in the block that follows the `def` statement ❷ is the body of the function. This code is executed when the function is called, not when the function is first defined.

The `hello()` lines after the function ❸ are function calls. In code, a function call is just the function's name followed by parentheses, possibly with some number of arguments in between the parentheses. When the program execution reaches these calls, it will jump to the top line in the function and begin executing the code there. When it reaches the end of the function, the execution returns to the line that called the function and continues moving through the code as before.

Since this program calls `hello()` three times, the code in the `hello()` function is executed three times. When you run this program, the output looks like this:

```
Howdy!
Howdy!!!
Hello there.
Howdy!
Howdy!!!
Hello there.
Howdy!
Howdy!!!
Hello there.
```

A major purpose of functions is to group code that gets executed multiple times. Without a function defined, you would have to copy and paste this code each time, and the program would look like this:

```
print('Howdy!')
print('Howdy!!!')
print('Hello there.')
print('Howdy!')
print('Howdy!!!')
print('Hello there.')
print('Howdy!')
print('Howdy!!!')
print('Hello there.')
```

In general, you always want to avoid duplicating code, because if you ever decide to update the code—if, for example, you find a bug you need to fix—you'll have to remember to change the code everywhere you copied it.

As you get more programming experience, you'll often find yourself *deduplicating* code, which means getting rid of duplicated or copy-and-pasted code. Deduplication makes your programs shorter, easier to read, and easier to update.

def Statements with Parameters

When you call the `print()` or `len()` function, you pass in values, called *arguments* in this context, by typing them between the parentheses. You can also define your own functions that accept arguments. Type this example into the file editor and save it as *helloFunc2.py*:

```
❶ def hello(name):
❷     print('Hello ' + name)

❸ hello('Alice')
  hello('Bob')
```

When you run this program, the output looks like this:

```
Hello Alice
Hello Bob
```

The definition of the `hello()` function in this program has a parameter called `name` ❶. A *parameter* is a variable that an argument is stored in when a function is called. The first time the `hello()` function is called, it's with the argument `'Alice'` ❸. The program execution enters the function, and the variable `name` is automatically set to `'Alice'`, which is what gets printed by the `print()` statement ❷.

One special thing to note about parameters is that the value stored in a parameter is forgotten when the function returns. For example, if you added `print(name)` after `hello('Bob')` in the previous program, the program would give you a `NameError` because there is no variable named `name`. This variable was destroyed after the function call `hello('Bob')` had returned, so `print(name)` would refer to a `name` variable that does not exist.

This is similar to how a program's variables are forgotten when the program terminates. I'll talk more about why that happens later in the chapter, when I discuss what a function's local scope is.

Return Values and return Statements

When you call the `len()` function and pass it an argument such as `'Hello'`, the function call evaluates to the integer value `5`, which is the length of the string you passed it. In general, the value that a function call evaluates to is called the *return value* of the function.

When creating a function using the `def` statement, you can specify what the return value should be with a `return` statement. A `return` statement consists of the following:

- The `return` keyword
- The value or expression that the function should return

When an expression is used with a `return` statement, the return value is what this expression evaluates to. For example, the following program defines a function that returns a different string depending on what number it is passed as an argument. Type this code into the file editor and save it as *magic8Ball.py*:

```
❶ import random

❷ def getAnswer(answerNumber):
❸     if answerNumber == 1:
          return 'It is certain'
      elif answerNumber == 2:
          return 'It is decidedly so'
      elif answerNumber == 3:
          return 'Yes'
      elif answerNumber == 4:
          return 'Reply hazy try again'
      elif answerNumber == 5:
          return 'Ask again later'
      elif answerNumber == 6:
          return 'Concentrate and ask again'
      elif answerNumber == 7:
          return 'My reply is no'
      elif answerNumber == 8:
          return 'Outlook not so good'
      elif answerNumber == 9:
          return 'Very doubtful'

❹ r = random.randint(1, 9)
❺ fortune = getAnswer(r)
❻ print(fortune)
```

When this program starts, Python first imports the `random` module ❶. Then the `getAnswer()` function is defined ❷. Because the function is being defined (and not called), the execution skips over the code in it. Next, the `random.randint()` function is called with two arguments, `1` and `9` ❹. It evaluates to a random integer between `1` and `9` (including `1` and `9` themselves), and this value is stored in a variable named `r`.

The `getAnswer()` function is called with `r` as the argument ❺. The program execution moves to the top of the `getAnswer()` function ❸, and the value `r` is stored in a parameter named `answerNumber`. Then, depending on this value in `answerNumber`, the function returns one of many possible string values. The program execution returns to the line at the bottom of the program that originally called `getAnswer()` ❺. The returned string is assigned to a variable named `fortune`, which then gets passed to a `print()` call ❻ and is printed to the screen.

Note that since you can pass return values as an argument to another function call, you could shorten these three lines:

```
r = random.randint(1, 9)
fortune = getAnswer(r)
print(fortune)
```

to this single equivalent line:

```
print(getAnswer(random.randint(1, 9)))
```

Remember, expressions are composed of values and operators. A function call can be used in an expression because it evaluates to its return value.

The None Value

In Python there is a value called `None`, which represents the absence of a value. `None` is the only value of the `NoneType` data type. (Other programming languages might call this value `null`, `nil`, or `undefined`.) Just like the Boolean `True` and `False` values, `None` must be typed with a capital *N*.

This value-without-a-value can be helpful when you need to store something that won't be confused for a real value in a variable. One place where `None` is used is as the return value of `print()`. The `print()` function displays text on the screen, but it doesn't need to return anything in the same way `len()` or `input()` does. But since all function calls need to evaluate to a return value, `print()` returns `None`. To see this in action, enter the following into the interactive shell:

```
>>> spam = print('Hello!')
Hello!
>>> None == spam
True
```

Behind the scenes, Python adds `return None` to the end of any function definition with no `return` statement. This is similar to how a `while` or `for` loop implicitly ends with a `continue` statement. Also, if you use a `return` statement without a value (that is, just the `return` keyword by itself), then `None` is returned.

Keyword Arguments and print()

Most arguments are identified by their position in the function call. For example, `random.randint(1, 10)` is different from `random.randint(10, 1)`. The function call `random.randint(1, 10)` will return a random integer between `1` and `10`, because the first argument is the low end of the range and the second argument is the high end (while `random.randint(10, 1)` causes an error).

However, *keyword arguments* are identified by the keyword put before them in the function call. Keyword arguments are often used for optional parameters. For example, the `print()` function has the optional parameters `end` and `sep` to specify what should be printed at the end of its arguments and between its arguments (separating them), respectively.

If you ran the following program:

```
print('Hello')
print('World')
```

the output would look like this:

```
Hello
World
```

The two strings appear on separate lines because the `print()` function automatically adds a newline character to the end of the string it is passed. However, you can set the `end` keyword argument to change this to a different string. For example, if the program were this:

```
print('Hello', end='')
print('World')
```

the output would look like this:

```
HelloWorld
```

The output is printed on a single line because there is no longer a newline printed after `'Hello'`. Instead, the blank string is printed. This is useful if you need to disable the newline that gets added to the end of every `print()` function call.

Similarly, when you pass multiple string values to `print()`, the function will automatically separate them with a single space. Enter the following into the interactive shell:

```
>>> print('cats', 'dogs', 'mice')
cats dogs mice
```

But you could replace the default separating string by passing the `sep` keyword argument. Enter the following into the interactive shell:

```
>>> print('cats', 'dogs', 'mice', sep=',')
cats,dogs,mice
```

You can add keyword arguments to the functions you write as well, but first you'll have to learn about the list and dictionary data types in the next two chapters. For now, just know that some functions have optional keyword arguments that can be specified when the function is called.

Local and Global Scope

Parameters and variables that are assigned in a called function are said to exist in that function's *local scope*. Variables that are assigned outside all functions are said to exist in the *global scope*. A variable that exists in a local scope is called a *local variable*, while a variable that exists in the global scope is called a *global variable*. A variable must be one or the other; it cannot be both local and global.

Think of a *scope* as a container for variables. When a scope is destroyed, all the values stored in the scope's variables are forgotten. There is only one global scope, and it is created when your program begins. When your program terminates, the global scope is destroyed, and all its variables are forgotten. Otherwise, the next time you ran your program, the variables would remember their values from the last time you ran it.

A local scope is created whenever a function is called. Any variables assigned in this function exist within the local scope. When the function returns, the local scope is destroyed, and these variables are forgotten. The next time you call this function, the local variables will not remember the values stored in them from the last time the function was called.

Scopes matter for several reasons:

- Code in the global scope cannot use any local variables.
- However, a local scope can access global variables.
- Code in a function's local scope cannot use variables in any other local scope.
- You can use the same name for different variables if they are in different scopes. That is, there can be a local variable named `spam` and a global variable also named `spam`.

The reason Python has different scopes instead of just making everything a global variable is so that when variables are modified by the code in a particular call to a function, the function interacts with the rest of the program only through its parameters and the return value. This narrows down the list code lines that may be causing a bug. If your program contained nothing but global variables and had a bug because of a variable being set to a bad value, then it would be hard to track down where this bad value was set. It could have been set from anywhere in the program—and your program could be hundreds or thousands of lines long! But if the bug is because of a local variable with a bad value, you know that only the code in that one function could have set it incorrectly.

While using global variables in small programs is fine, it is a bad habit to rely on global variables as your programs get larger and larger.

Local Variables Cannot Be Used in the Global Scope

Consider this program, which will cause an error when you run it:

```
def spam():
    eggs = 31337
```

```
spam()
print(eggs)
```

If you run this program, the output will look like this:

```
Traceback (most recent call last):
  File "C:/test3784.py", line 4, in <module>
    print(eggs)
NameError: name 'eggs' is not defined
```

The error happens because the eggs variable exists only in the local scope created when spam() is called. Once the program execution returns from spam, that local scope is destroyed, and there is no longer a variable named eggs. So when your program tries to run print(eggs), Python gives you an error saying that eggs is not defined. This makes sense if you think about it; when the program execution is in the global scope, no local scopes exist, so there can't be any local variables. This is why only global variables can be used in the global scope.

Local Scopes Cannot Use Variables in Other Local Scopes

A new local scope is created whenever a function is called, including when a function is called from another function. Consider this program:

```
  def spam():
❶     eggs = 99
❷     bacon()
❸     print(eggs)

  def bacon():
      ham = 101
❹     eggs = 0

❺ spam()
```

When the program starts, the spam() function is called ❺, and a local scope is created. The local variable eggs ❶ is set to 99. Then the bacon() function is called ❷, and a second local scope is created. Multiple local scopes can exist at the same time. In this new local scope, the local variable ham is set to 101, and a local variable eggs—which is different from the one in spam()'s local scope—is also created ❹ and set to 0.

When bacon() returns, the local scope for that call is destroyed. The program execution continues in the spam() function to print the value of eggs ❸, and since the local scope for the call to spam() still exists here, the eggs variable is set to 99. This is what the program prints.

The upshot is that local variables in one function are completely separate from the local variables in another function.

Global Variables Can Be Read from a Local Scope

Consider the following program:

```
def spam():
    print(eggs)
eggs = 42
spam()
print(eggs)
```

Since there is no parameter named eggs or any code that assigns eggs a value in the spam() function, when eggs is used in spam(), Python considers it a reference to the global variable eggs. This is why 42 is printed when the previous program is run.

Local and Global Variables with the Same Name

To simplify your life, avoid using local variables that have the same name as a global variable or another local variable. But technically, it's perfectly legal to do so in Python. To see what happens, type the following code into the file editor and save it as *sameName.py*:

```
  def spam():
❶     eggs = 'spam local'
      print(eggs)    # prints 'spam local'

  def bacon():
❷     eggs = 'bacon local'
      print(eggs)    # prints 'bacon local'
      spam()
      print(eggs)    # prints 'bacon local'

❸ eggs = 'global'
  bacon()
  print(eggs)        # prints 'global'
```

When you run this program, it outputs the following:

```
bacon local
spam local
bacon local
global
```

There are actually three different variables in this program, but confusingly they are all named eggs. The variables are as follows:

❶ A variable named eggs that exists in a local scope when spam() is called.

❷ A variable named eggs that exists in a local scope when bacon() is called.

❸ A variable named eggs that exists in the global scope.

Since these three separate variables all have the same name, it can be confusing to keep track of which one is being used at any given time. This is why you should avoid using the same variable name in different scopes.

The global Statement

If you need to modify a global variable from within a function, use the `global` statement. If you have a line such as `global eggs` at the top of a function, it tells Python, "In this function, `eggs` refers to the global variable, so don't create a local variable with this name." For example, type the following code into the file editor and save it as *sameName2.py*:

```
def spam():
❶   global eggs
❷   eggs = 'spam'

eggs = 'global'
spam()
print(eggs)
```

When you run this program, the final `print()` call will output this:

```
spam
```

Because `eggs` is declared `global` at the top of `spam()` ❶, when `eggs` is set to `'spam'` ❷, this assignment is done to the globally scoped `spam`. No local `spam` variable is created.

There are four rules to tell whether a variable is in a local scope or global scope:

1. If a variable is being used in the global scope (that is, outside of all functions), then it is always a global variable.
2. If there is a `global` statement for that variable in a function, it is a global variable.
3. Otherwise, if the variable is used in an assignment statement in the function, it is a local variable.
4. But if the variable is not used in an assignment statement, it is a global variable.

To get a better feel for these rules, here's an example program. Type the following code into the file editor and save it as *sameName3.py*:

```
  def spam():
❶   global eggs
    eggs = 'spam' # this is the global

  def bacon():
❷   eggs = 'bacon' # this is a local
```

```
def ham():
❸   print(eggs) # this is the global

eggs = 42 # this is the global
spam()
print(eggs)
```

In the spam() function, eggs is the global eggs variable, because there's a global statement for eggs at the beginning of the function ❶. In bacon(), eggs is a local variable, because there's an assignment statement for it in that function ❷. In ham() ❸, eggs is the global variable, because there is no assignment statement or global statement for it in that function. If you run *sameName3.py*, the output will look like this:

```
spam
```

In a function, a variable will either always be global or always be local. There's no way that the code in a function can use a local variable named eggs and then later in that same function use the global eggs variable.

NOTE *If you ever want to modify the value stored in a global variable from in a function, you must use a* global *statement on that variable.*

If you try to use a local variable in a function before you assign a value to it, as in the following program, Python will give you an error. To see this, type the following into the file editor and save it as *sameName4.py*:

```
def spam():
    print(eggs) # ERROR!
❶   eggs = 'spam local'

❷ eggs = 'global'
spam()
```

If you run the previous program, it produces an error message.

```
Traceback (most recent call last):
  File "C:/test3784.py", line 6, in <module>
    spam()
  File "C:/test3784.py", line 2, in spam
    print(eggs) # ERROR!
UnboundLocalError: local variable 'eggs' referenced before assignment
```

This error happens because Python sees that there is an assignment statement for eggs in the spam() function ❶ and therefore considers eggs to be local. But because print(eggs) is executed before eggs is assigned anything, the local variable eggs doesn't exist. Python will *not* fall back to using the global eggs variable ❷.

> **FUNCTIONS AS "BLACK BOXES"**
>
> Often, all you need to know about a function are its inputs (the parameters) and output value; you don't always have to burden yourself with how the function's code actually works. When you think about functions in this high-level way, it's common to say that you're treating the function as a "black box."
>
> This idea is fundamental to modern programming. Later chapters in this book will show you several modules with functions that were written by other people. While you can take a peek at the source code if you're curious, you don't need to know how these functions work in order to use them. And because writing functions without global variables is encouraged, you usually don't have to worry about the function's code interacting with the rest of your program.

Exception Handling

Right now, getting an error, or *exception*, in your Python program means the entire program will crash. You don't want this to happen in real-world programs. Instead, you want the program to detect errors, handle them, and then continue to run.

For example, consider the following program, which has a "divide-by-zero" error. Open a new file editor window and enter the following code, saving it as *zeroDivide.py*:

```
def spam(divideBy):
    return 42 / divideBy

print(spam(2))
print(spam(12))
print(spam(0))
print(spam(1))
```

We've defined a function called `spam`, given it a parameter, and then printed the value of that function with various parameters to see what happens. This is the output you get when you run the previous code:

```
21.0
3.5
Traceback (most recent call last):
  File "C:/zeroDivide.py", line 6, in <module>
    print(spam(0))
  File "C:/zeroDivide.py", line 2, in spam
    return 42 / divideBy
ZeroDivisionError: division by zero
```

A `ZeroDivisionError` happens whenever you try to divide a number by zero. From the line number given in the error message, you know that the `return` statement in `spam()` is causing an error.

Errors can be handled with `try` and `except` statements. The code that could potentially have an error is put in a `try` clause. The program execution moves to the start of a following `except` clause if an error happens.

You can put the previous divide-by-zero code in a `try` clause and have an `except` clause contain code to handle what happens when this error occurs.

```
def spam(divideBy):
    try:
        return 42 / divideBy
    except ZeroDivisionError:
        print('Error: Invalid argument.')

print(spam(2))
print(spam(12))
print(spam(0))
print(spam(1))
```

When code in a `try` clause causes an error, the program execution immediately moves to the code in the `except` clause. After running that code, the execution continues as normal. The output of the previous program is as follows:

```
21.0
3.5
Error: Invalid argument.
None
42.0
```

Note that any errors that occur in function calls in a `try` block will also be caught. Consider the following program, which instead has the `spam()` calls in the `try` block:

```
def spam(divideBy):
    return 42 / divideBy

try:
    print(spam(2))
    print(spam(12))
    print(spam(0))
    print(spam(1))
except ZeroDivisionError:
    print('Error: Invalid argument.')
```

When this program is run, the output looks like this:

```
21.0
3.5
Error: Invalid argument.
```

The reason `print(spam(1))` is never executed is because once the execution jumps to the code in the `except` clause, it does not return to the `try` clause. Instead, it just continues moving down as normal.

A Short Program: Guess the Number

The toy examples I've show you so far are useful for introducing basic concepts, but now let's see how everything you've learned comes together in a more complete program. In this section, I'll show you a simple "guess the number" game. When you run this program, the output will look something like this:

```
I am thinking of a number between 1 and 20.
Take a guess.
10
Your guess is too low.
Take a guess.
15
Your guess is too low.
Take a guess.
17
Your guess is too high.
Take a guess.
16
Good job! You guessed my number in 4 guesses!
```

Type the following source code into the file editor, and save the file as *guessTheNumber.py*:

```
# This is a guess the number game.
import random
secretNumber = random.randint(1, 20)
print('I am thinking of a number between 1 and 20.')

# Ask the player to guess 6 times.
for guessesTaken in range(1, 7):
    print('Take a guess.')
    guess = int(input())

    if guess < secretNumber:
        print('Your guess is too low.')
    elif guess > secretNumber:
        print('Your guess is too high.')
    else:
        break    # This condition is the correct guess!

if guess == secretNumber:
    print('Good job! You guessed my number in ' + str(guessesTaken) + ' guesses!')
else:
    print('Nope. The number I was thinking of was ' + str(secretNumber))
```

Let's look at this code line by line, starting at the top.

```
# This is a guess the number game.
import random
secretNumber = random.randint(1, 20)
```

First, a comment at the top of the code explains what the program does. Then, the program imports the `random` module so that it can use the `random.randint()` function to generate a number for the user to guess. The return value, a random integer between 1 and 20, is stored in the variable `secretNumber`.

```
print('I am thinking of a number between 1 and 20.')

# Ask the player to guess 6 times.
for guessesTaken in range(1, 7):
    print('Take a guess.')
    guess = int(input())
```

The program tells the player that it has come up with a secret number and will give the player six chances to guess it. The code that lets the player enter a guess and checks that guess is in a `for` loop that will loop at most six times. The first thing that happens in the loop is that the player types in a guess. Since `input()` returns a string, its return value is passed straight into `int()`, which translates the string into an integer value. This gets stored in a variable named `guess`.

```
    if guess < secretNumber:
        print('Your guess is too low.')
    elif guess > secretNumber:
        print('Your guess is too high.')
```

These few lines of code check to see whether the guess is less than or greater than the secret number. In either case, a hint is printed to the screen.

```
    else:
        break    # This condition is the correct guess!
```

If the guess is neither higher nor lower than the secret number, then it must be equal to the secret number, in which case you want the program execution to break out of the `for` loop.

```
if guess == secretNumber:
    print('Good job! You guessed my number in ' + str(guessesTaken) + ' guesses!')
else:
    print('Nope. The number I was thinking of was ' + str(secretNumber))
```

After the `for` loop, the previous `if...else` statement checks whether the player has correctly guessed the number and prints an appropriate message to the screen. In both cases, the program displays a variable that contains

an integer value (`guessesTaken` and `secretNumber`). Since it must concatenate these integer values to strings, it passes these variables to the `str()` function, which returns the string value form of these integers. Now these strings can be concatenated with the `+` operators before finally being passed to the `print()` function call.

Summary

Functions are the primary way to compartmentalize your code into logical groups. Since the variables in functions exist in their own local scopes, the code in one function cannot directly affect the values of variables in other functions. This limits what code could be changing the values of your variables, which can be helpful when it comes to debugging your code.

Functions are a great tool to help you organize your code. You can think of them as black boxes: They have inputs in the form of parameters and outputs in the form of return values, and the code in them doesn't affect variables in other functions.

In previous chapters, a single error could cause your programs to crash. In this chapter, you learned about `try` and `except` statements, which can run code when an error has been detected. This can make your programs more resilient to common error cases.

Practice Questions

1. Why are functions advantageous to have in your programs?
2. When does the code in a function execute: when the function is defined or when the function is called?
3. What statement creates a function?
4. What is the difference between a function and a function call?
5. How many global scopes are there in a Python program? How many local scopes?
6. What happens to variables in a local scope when the function call returns?
7. What is a return value? Can a return value be part of an expression?
8. If a function does not have a return statement, what is the return value of a call to that function?
9. How can you force a variable in a function to refer to the global variable?
10. What is the data type of `None`?
11. What does the `import areallyourpetsnamederic` statement do?
12. If you had a function named `bacon()` in a module named `spam`, how would you call it after importing `spam`?
13. How can you prevent a program from crashing when it gets an error?
14. What goes in the `try` clause? What goes in the `except` clause?

Practice Projects

For practice, write programs to do the following tasks.

The Collatz Sequence

Write a function named `collatz()` that has one parameter named `number`. If `number` is even, then `collatz()` should print `number // 2` and return this value. If `number` is odd, then `collatz()` should print and return `3 * number + 1`.

Then write a program that lets the user type in an integer and that keeps calling `collatz()` on that number until the function returns the value `1`. (Amazingly enough, this sequence actually works for any integer—sooner or later, using this sequence, you'll arrive at 1! Even mathematicians aren't sure why. Your program is exploring what's called the *Collatz sequence*, sometimes called "the simplest impossible math problem.")

Remember to convert the return value from `input()` to an integer with the `int()` function; otherwise, it will be a string value.

Hint: An integer `number` is even if `number % 2 == 0`, and it's odd if `number % 2 == 1`.

The output of this program could look something like this:

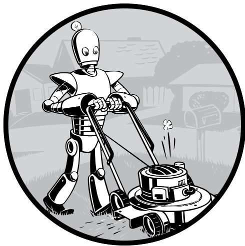
```
Enter number:
3
10
5
16
8
4
2
1
```

Input Validation

Add `try` and `except` statements to the previous project to detect whether the user types in a noninteger string. Normally, the `int()` function will raise a `ValueError` error if it is passed a noninteger string, as in `int('puppy')`. In the `except` clause, print a message to the user saying they must enter an integer.

4

LISTS



One more topic you'll need to understand before you can begin writing programs in earnest is the list data type and its cousin, the tuple. Lists and tuples can contain multiple values, which makes it easier to write programs that handle large amounts of data. And since lists themselves can contain other lists, you can use them to arrange data into hierarchical structures.

In this chapter, I'll discuss the basics of lists. I'll also teach you about methods, which are functions that are tied to values of a certain data type. Then I'll briefly cover the list-like tuple and string data types and how they compare to list values. In the next chapter, I'll introduce you to the dictionary data type.

The List Data Type

A *list* is a value that contains multiple values in an ordered sequence. The term *list value* refers to the list itself (which is a value that can be stored in a variable or passed to a function like any other value), not the values inside the list value. A list value looks like this: `['cat', 'bat', 'rat', 'elephant']`. Just as string values are typed with quote characters to mark where the string begins and ends, a list begins with an opening square bracket and ends with a closing square bracket, `[]`. Values inside the list are also called *items*. Items are separated with commas (that is, they are *comma-delimited*). For example, enter the following into the interactive shell:

```
>>> [1, 2, 3]
[1, 2, 3]
>>> ['cat', 'bat', 'rat', 'elephant']
['cat', 'bat', 'rat', 'elephant']
>>> ['hello', 3.1415, True, None, 42]
['hello', 3.1415, True, None, 42]
❶ >>> spam = ['cat', 'bat', 'rat', 'elephant']
>>> spam
['cat', 'bat', 'rat', 'elephant']
```

The `spam` variable ❶ is still assigned only one value: the list value. But the list value itself contains other values. The value `[]` is an empty list that contains no values, similar to `''`, the empty string.

Getting Individual Values in a List with Indexes

Say you have the list `['cat', 'bat', 'rat', 'elephant']` stored in a variable named `spam`. The Python code `spam[0]` would evaluate to `'cat'`, and `spam[1]` would evaluate to `'bat'`, and so on. The integer inside the square brackets that follows the list is called an *index*. The first value in the list is at index `0`, the second value is at index `1`, the third value is at index `2`, and so on. Figure 4-1 shows a list value assigned to `spam`, along with what the index expressions would evaluate to.

```
spam = ["cat", "bat", "rat", "elephant"]

spam[0]   spam[1]   spam[2]   spam[3]
```

Figure 4-1: A list value stored in the variable spam, showing which value each index refers to

For example, type the following expressions into the interactive shell. Start by assigning a list to the variable `spam`.

```
>>> spam = ['cat', 'bat', 'rat', 'elephant']
>>> spam[0]
'cat'
>>> spam[1]
'bat'
>>> spam[2]
'rat'
>>> spam[3]
'elephant'
```

```
>>> ['cat', 'bat', 'rat', 'elephant'][3]
'elephant'
❶ >>> 'Hello ' + spam[0]
❷ 'Hello cat'
>>> 'The ' + spam[1] + ' ate the ' + spam[0] + '.'
'The bat ate the cat.'
```

Notice that the expression `'Hello ' + spam[0]` ❶ evaluates to `'Hello ' + 'cat'` because `spam[0]` evaluates to the string `'cat'`. This expression in turn evaluates to the string value `'Hello cat'` ❷.

Python will give you an `IndexError` error message if you use an index that exceeds the number of values in your list value.

```
>>> spam = ['cat', 'bat', 'rat', 'elephant']
>>> spam[10000]
Traceback (most recent call last):
  File "<pyshell#9>", line 1, in <module>
    spam[10000]
IndexError: list index out of range
```

Indexes can be only integer values, not floats. The following example will cause a `TypeError` error:

```
>>> spam = ['cat', 'bat', 'rat', 'elephant']
>>> spam[1]
'bat'
>>> spam[1.0]
Traceback (most recent call last):
  File "<pyshell#13>", line 1, in <module>
    spam[1.0]
TypeError: list indices must be integers, not float
>>> spam[int(1.0)]
'bat'
```

Lists can also contain other list values. The values in these lists of lists can be accessed using multiple indexes, like so:

```
>>> spam = [['cat', 'bat'], [10, 20, 30, 40, 50]]
>>> spam[0]
['cat', 'bat']
>>> spam[0][1]
'bat'
>>> spam[1][4]
50
```

The first index dictates which list value to use, and the second indicates the value within the list value. For example, `spam[0][1]` prints `'bat'`, the second value in the first list. If you only use one index, the program will print the full list value at that index.

Negative Indexes

While indexes start at 0 and go up, you can also use negative integers for the index. The integer value -1 refers to the last index in a list, the value -2 refers to the second-to-last index in a list, and so on. Enter the following into the interactive shell:

```
>>> spam = ['cat', 'bat', 'rat', 'elephant']
>>> spam[-1]
'elephant'
>>> spam[-3]
'bat'
>>> 'The ' + spam[-1] + ' is afraid of the ' + spam[-3] + '.'
'The elephant is afraid of the bat.'
```

Getting Sublists with Slices

Just as an index can get a single value from a list, a *slice* can get several values from a list, in the form of a new list. A slice is typed between square brackets, like an index, but it has two integers separated by a colon. Notice the difference between indexes and slices.

- spam[2] is a list with an index (one integer).
- spam[1:4] is a list with a slice (two integers).

In a slice, the first integer is the index where the slice starts. The second integer is the index where the slice ends. A slice goes up to, but will not include, the value at the second index. A slice evaluates to a new list value. Enter the following into the interactive shell:

```
>>> spam = ['cat', 'bat', 'rat', 'elephant']
>>> spam[0:4]
['cat', 'bat', 'rat', 'elephant']
>>> spam[1:3]
['bat', 'rat']
>>> spam[0:-1]
['cat', 'bat', 'rat']
```

As a shortcut, you can leave out one or both of the indexes on either side of the colon in the slice. Leaving out the first index is the same as using 0, or the beginning of the list. Leaving out the second index is the same as using the length of the list, which will slice to the end of the list. Enter the following into the interactive shell:

```
>>> spam = ['cat', 'bat', 'rat', 'elephant']
>>> spam[:2]
['cat', 'bat']
>>> spam[1:]
['bat', 'rat', 'elephant']
```

```
>>> spam[:]
['cat', 'bat', 'rat', 'elephant']
```

Getting a List's Length with len()

The `len()` function will return the number of values that are in a list value passed to it, just like it can count the number of characters in a string value. Enter the following into the interactive shell:

```
>>> spam = ['cat', 'dog', 'moose']
>>> len(spam)
3
```

Changing Values in a List with Indexes

Normally a variable name goes on the left side of an assignment statement, like `spam = 42`. However, you can also use an index of a list to change the value at that index. For example, `spam[1] = 'aardvark'` means "Assign the value at index `1` in the list `spam` to the string `'aardvark'`." Enter the following into the interactive shell:

```
>>> spam = ['cat', 'bat', 'rat', 'elephant']
>>> spam[1] = 'aardvark'
>>> spam
['cat', 'aardvark', 'rat', 'elephant']
>>> spam[2] = spam[1]
>>> spam
['cat', 'aardvark', 'aardvark', 'elephant']
>>> spam[-1] = 12345
>>> spam
['cat', 'aardvark', 'aardvark', 12345]
```

List Concatenation and List Replication

The `+` operator can combine two lists to create a new list value in the same way it combines two strings into a new string value. The `*` operator can also be used with a list and an integer value to replicate the list. Enter the following into the interactive shell:

```
>>> [1, 2, 3] + ['A', 'B', 'C']
[1, 2, 3, 'A', 'B', 'C']
>>> ['X', 'Y', 'Z'] * 3
['X', 'Y', 'Z', 'X', 'Y', 'Z', 'X', 'Y', 'Z']
>>> spam = [1, 2, 3]
>>> spam = spam + ['A', 'B', 'C']
>>> spam
[1, 2, 3, 'A', 'B', 'C']
```

Removing Values from Lists with del Statements

The `del` statement will delete values at an index in a list. All of the values in the list after the deleted value will be moved up one index. For example, enter the following into the interactive shell:

```
>>> spam = ['cat', 'bat', 'rat', 'elephant']
>>> del spam[2]
>>> spam
['cat', 'bat', 'elephant']
>>> del spam[2]
>>> spam
['cat', 'bat']
```

The `del` statement can also be used on a simple variable to delete it, as if it were an "unassignment" statement. If you try to use the variable after deleting it, you will get a `NameError` error because the variable no longer exists.

In practice, you almost never need to delete simple variables. The `del` statement is mostly used to delete values from lists.

Working with Lists

When you first begin writing programs, it's tempting to create many individual variables to store a group of similar values. For example, if I wanted to store the names of my cats, I might be tempted to write code like this:

```
catName1 = 'Zophie'
catName2 = 'Pooka'
catName3 = 'Simon'
catName4 = 'Lady Macbeth'
catName5 = 'Fat-tail'
catName6 = 'Miss Cleo'
```

(I don't actually own this many cats, I swear.) It turns out that this is a bad way to write code. For one thing, if the number of cats changes, your program will never be able to store more cats than you have variables. These types of programs also have a lot of duplicate or nearly identical code in them. Consider how much duplicate code is in the following program, which you should enter into the file editor and save as *allMyCats1.py*:

```
print('Enter the name of cat 1:')
catName1 = input()
print('Enter the name of cat 2:')
catName2 = input()
print('Enter the name of cat 3:')
catName3 = input()
print('Enter the name of cat 4:')
catName4 = input()
print('Enter the name of cat 5:')
catName5 = input()
```

```
print('Enter the name of cat 6:')
catName6 = input()
print('The cat names are:')
print(catName1 + ' ' + catName2 + ' ' + catName3 + ' ' + catName4 + ' ' +
catName5 + ' ' + catName6)
```

Instead of using multiple, repetitive variables, you can use a single variable that contains a list value. For example, here's a new and improved version of the *allMyCats1.py* program. This new version uses a single list and can store any number of cats that the user types in. In a new file editor window, type the following source code and save it as *allMyCats2.py*:

```
catNames = []
while True:
    print('Enter the name of cat ' + str(len(catNames) + 1) +
      ' (Or enter nothing to stop.):')
    name = input()
    if name == '':
        break
    catNames = catNames + [name]  # list concatenation
print('The cat names are:')
for name in catNames:
    print('  ' + name)
```

When you run this program, the output will look something like this:

```
Enter the name of cat 1 (Or enter nothing to stop.):
Zophie
Enter the name of cat 2 (Or enter nothing to stop.):
Pooka
Enter the name of cat 3 (Or enter nothing to stop.):
Simon
Enter the name of cat 4 (Or enter nothing to stop.):
Lady Macbeth
Enter the name of cat 5 (Or enter nothing to stop.):
Fat-tail
Enter the name of cat 6 (Or enter nothing to stop.):
Miss Cleo
Enter the name of cat 7 (Or enter nothing to stop.):

The cat names are:
  Zophie
  Pooka
  Simon
  Lady Macbeth
  Fat-tail
  Miss Cleo
```

The benefit of using a list is that your data is now in a structure, so your program is much more flexible in processing the data than it would be with several repetitive variables.

Using for Loops with Lists

In Chapter 2, you learned about using `for` loops to execute a block of code a certain number of times. Technically, a `for` loop repeats the code block once for each value in a list or list-like value. For example, if you ran this code:

```
for i in range(4):
    print(i)
```

the output of this program would be as follows:

```
0
1
2
3
```

This is because the return value from `range(4)` is a list-like value that Python considers similar to `[0, 1, 2, 3]`. The following program has the same output as the previous one:

```
for i in [0, 1, 2, 3]:
    print(i)
```

What the previous `for` loop actually does is loop through its clause with the variable `i` set to a successive value in the `[0, 1, 2, 3]` list in each iteration.

NOTE *In this book, I use the term* list-like *to refer to data types that are technically named* sequences. *You don't need to know the technical definitions of this term, though.*

A common Python technique is to use `range(len(`*`someList`*`))` with a `for` loop to iterate over the indexes of a list. For example, enter the following into the interactive shell:

```
>>> supplies = ['pens', 'staplers', 'flame-throwers', 'binders']
>>> for i in range(len(supplies)):
    print('Index ' + str(i) + ' in supplies is: ' + supplies[i])

Index 0 in supplies is: pens
Index 1 in supplies is: staplers
Index 2 in supplies is: flame-throwers
Index 3 in supplies is: binders
```

Using `range(len(supplies))` in the previously shown `for` loop is handy because the code in the loop can access the index (as the variable `i`) and the value at that index (as `supplies[i]`). Best of all, `range(len(supplies))` will iterate through all the indexes of `supplies`, no matter how many items it contains.

The in and not in Operators

You can determine whether a value is or isn't in a list with the `in` and `not in` operators. Like other operators, `in` and `not in` are used in expressions and connect two values: a value to look for in a list and the list where it may be found. These expressions will evaluate to a Boolean value. Enter the following into the interactive shell:

```
>>> 'howdy' in ['hello', 'hi', 'howdy', 'heyas']
True
>>> spam = ['hello', 'hi', 'howdy', 'heyas']
>>> 'cat' in spam
False
>>> 'howdy' not in spam
False
>>> 'cat' not in spam
True
```

For example, the following program lets the user type in a pet name and then checks to see whether the name is in a list of pets. Open a new file editor window, enter the following code, and save it as *myPets.py*:

```
myPets = ['Zophie', 'Pooka', 'Fat-tail']
print('Enter a pet name:')
name = input()
if name not in myPets:
    print('I do not have a pet named ' + name)
else:
    print(name + ' is my pet.')
```

The output may look something like this:

```
Enter a pet name:
Footfoot
I do not have a pet named Footfoot
```

The Multiple Assignment Trick

The *multiple assignment trick* is a shortcut that lets you assign multiple variables with the values in a list in one line of code. So instead of doing this:

```
>>> cat = ['fat', 'black', 'loud']
>>> size = cat[0]
>>> color = cat[1]
>>> disposition = cat[2]
```

you could type this line of code:

```
>>> cat = ['fat', 'black', 'loud']
>>> size, color, disposition = cat
```

The number of variables and the length of the list must be exactly equal, or Python will give you a ValueError:

```
>>> cat = ['fat', 'black', 'loud']
>>> size, color, disposition, name = cat
Traceback (most recent call last):
  File "<pyshell#84>", line 1, in <module>
    size, color, disposition, name = cat
ValueError: need more than 3 values to unpack
```

Augmented Assignment Operators

When assigning a value to a variable, you will frequently use the variable itself. For example, after assigning 42 to the variable spam, you would increase the value in spam by 1 with the following code:

```
>>> spam = 42
>>> spam = spam + 1
>>> spam
43
```

As a shortcut, you can use the augmented assignment operator += to do the same thing:

```
>>> spam = 42
>>> spam += 1
>>> spam
43
```

There are augmented assignment operators for the +, -, *, /, and % operators, described in Table 4-1.

Table 4-1: The Augmented Assignment Operators

Augmented assignment statement	Equivalent assignment statement
spam = spam + 1	spam += 1
spam = spam - 1	spam -= 1
spam = spam * 1	spam *= 1
spam = spam / 1	spam /= 1
spam = spam % 1	spam %= 1

The += operator can also do string and list concatenation, and the *= operator can do string and list replication. Enter the following into the interactive shell:

```
>>> spam = 'Hello'
>>> spam += ' world!'
>>> spam
'Hello world!'
```

```
>>> bacon = ['Zophie']
>>> bacon *= 3
>>> bacon
['Zophie', 'Zophie', 'Zophie']
```

Methods

A *method* is the same thing as a function, except it is "called on" a value. For example, if a list value were stored in `spam`, you would call the `index()` list method (which I'll explain next) on that list like so: `spam.index('hello')`. The method part comes after the value, separated by a period.

Each data type has its own set of methods. The list data type, for example, has several useful methods for finding, adding, removing, and otherwise manipulating values in a list.

Finding a Value in a List with the index() Method

List values have an `index()` method that can be passed a value, and if that value exists in the list, the index of the value is returned. If the value isn't in the list, then Python produces a `ValueError` error. Enter the following into the interactive shell:

```
>>> spam = ['hello', 'hi', 'howdy', 'heyas']
>>> spam.index('hello')
0
>>> spam.index('heyas')
3
>>> spam.index('howdy howdy howdy')
Traceback (most recent call last):
  File "<pyshell#31>", line 1, in <module>
    spam.index('howdy howdy howdy')
ValueError: 'howdy howdy howdy' is not in list
```

When there are duplicates of the value in the list, the index of its first appearance is returned. Enter the following into the interactive shell, and notice that `index()` returns `1`, not `3`:

```
>>> spam = ['Zophie', 'Pooka', 'Fat-tail', 'Pooka']
>>> spam.index('Pooka')
1
```

Adding Values to Lists with the append() and insert() Methods

To add new values to a list, use the `append()` and `insert()` methods. Enter the following into the interactive shell to call the `append()` method on a list value stored in the variable `spam`:

```
>>> spam = ['cat', 'dog', 'bat']
>>> spam.append('moose')
```

```
>>> spam
['cat', 'dog', 'bat', 'moose']
```

The previous `append()` method call adds the argument to the end of the list. The `insert()` method can insert a value at any index in the list. The first argument to `insert()` is the index for the new value, and the second argument is the new value to be inserted. Enter the following into the interactive shell:

```
>>> spam = ['cat', 'dog', 'bat']
>>> spam.insert(1, 'chicken')
>>> spam
['cat', 'chicken', 'dog', 'bat']
```

Notice that the code is `spam.append('moose')` and `spam.insert(1, 'chicken')`, not `spam = spam.append('moose')` and `spam = spam.insert(1, 'chicken')`. Neither `append()` nor `insert()` gives the new value of `spam` as its return value. (In fact, the return value of `append()` and `insert()` is `None`, so you definitely wouldn't want to store this as the new variable value.) Rather, the list is modified *in place*. Modifying a list in place is covered in more detail later in "Mutable and Immutable Data Types" on page 94.

Methods belong to a single data type. The `append()` and `insert()` methods are list methods and can be called only on list values, not on other values such as strings or integers. Enter the following into the interactive shell, and note the `AttributeError` error messages that show up:

```
>>> eggs = 'hello'
>>> eggs.append('world')
Traceback (most recent call last):
  File "<pyshell#19>", line 1, in <module>
    eggs.append('world')
AttributeError: 'str' object has no attribute 'append'
>>> bacon = 42
>>> bacon.insert(1, 'world')
Traceback (most recent call last):
  File "<pyshell#22>", line 1, in <module>
    bacon.insert(1, 'world')
AttributeError: 'int' object has no attribute 'insert'
```

Removing Values from Lists with remove()

The `remove()` method is passed the value to be removed from the list it is called on. Enter the following into the interactive shell:

```
>>> spam = ['cat', 'bat', 'rat', 'elephant']
>>> spam.remove('bat')
>>> spam
['cat', 'rat', 'elephant']
```

Attempting to delete a value that does not exist in the list will result in a `ValueError` error. For example, enter the following into the interactive shell and notice the error that is displayed:

```
>>> spam = ['cat', 'bat', 'rat', 'elephant']
>>> spam.remove('chicken')
Traceback (most recent call last):
  File "<pyshell#11>", line 1, in <module>
    spam.remove('chicken')
ValueError: list.remove(x): x not in list
```

If the value appears multiple times in the list, only the first instance of the value will be removed. Enter the following into the interactive shell:

```
>>> spam = ['cat', 'bat', 'rat', 'cat', 'hat', 'cat']
>>> spam.remove('cat')
>>> spam
['bat', 'rat', 'cat', 'hat', 'cat']
```

The `del` statement is good to use when you know the index of the value you want to remove from the list. The `remove()` method is good when you know the value you want to remove from the list.

Sorting the Values in a List with the sort() Method

Lists of number values or lists of strings can be sorted with the `sort()` method. For example, enter the following into the interactive shell:

```
>>> spam = [2, 5, 3.14, 1, -7]
>>> spam.sort()
>>> spam
[-7, 1, 2, 3.14, 5]
>>> spam = ['ants', 'cats', 'dogs', 'badgers', 'elephants']
>>> spam.sort()
>>> spam
['ants', 'badgers', 'cats', 'dogs', 'elephants']
```

You can also pass `True` for the `reverse` keyword argument to have `sort()` sort the values in reverse order. Enter the following into the interactive shell:

```
>>> spam.sort(reverse=True)
>>> spam
['elephants', 'dogs', 'cats', 'badgers', 'ants']
```

There are three things you should note about the `sort()` method. First, the `sort()` method sorts the list in place; don't try to capture the return value by writing code like `spam = spam.sort()`.

Second, you cannot sort lists that have both number values *and* string values in them, since Python doesn't know how to compare these values. Type the following into the interactive shell and notice the `TypeError` error:

```
>>> spam = [1, 3, 2, 4, 'Alice', 'Bob']
>>> spam.sort()
Traceback (most recent call last):
  File "<pyshell#70>", line 1, in <module>
    spam.sort()
TypeError: unorderable types: str() < int()
```

Third, `sort()` uses "ASCIIbetical order" rather than actual alphabetical order for sorting strings. This means uppercase letters come before lowercase letters. Therefore, the lowercase *a* is sorted so that it comes *after* the uppercase *Z*. For an example, enter the following into the interactive shell:

```
>>> spam = ['Alice', 'ants', 'Bob', 'badgers', 'Carol', 'cats']
>>> spam.sort()
>>> spam
['Alice', 'Bob', 'Carol', 'ants', 'badgers', 'cats']
```

If you need to sort the values in regular alphabetical order, pass `str.lower` for the `key` keyword argument in the `sort()` method call.

```
>>> spam = ['a', 'z', 'A', 'Z']
>>> spam.sort(key=str.lower)
>>> spam
['a', 'A', 'z', 'Z']
```

This causes the `sort()` function to treat all the items in the list as if they were lowercase without actually changing the values in the list.

Example Program: Magic 8 Ball with a List

Using lists, you can write a much more elegant version of the previous chapter's Magic 8 Ball program. Instead of several lines of nearly identical `elif` statements, you can create a single list that the code works with. Open a new file editor window and enter the following code. Save it as *magic8Ball2.py*.

```
import random

messages = ['It is certain',
    'It is decidedly so',
    'Yes definitely',
    'Reply hazy try again',
    'Ask again later',
    'Concentrate and ask again',
    'My reply is no',
    'Outlook not so good',
    'Very doubtful']

print(messages[random.randint(0, len(messages) - 1)])
```

EXCEPTIONS TO INDENTATION RULES IN PYTHON

In most cases, the amount of indentation for a line of code tells Python what block it is in. There are some exceptions to this rule, however. For example, lists can actually span several lines in the source code file. The indentation of these lines do not matter; Python knows that until it sees the ending square bracket, the list is not finished. For example, you can have code that looks like this:

```
spam = ['apples',
    'oranges',
                    'bananas',
'cats']
print(spam)
```

Of course, practically speaking, most people use Python's behavior to make their lists look pretty and readable, like the messages list in the Magic 8 Ball program.

You can also split up a single instruction across multiple lines using the \ line continuation character at the end. Think of \ as saying, "This instruction continues on the next line." The indentation on the line after a \ line continuation is not significant. For example, the following is valid Python code:

```
print('Four score and seven ' + \
      'years ago...')
```

These tricks are useful when you want to rearrange long lines of Python code to be a bit more readable.

When you run this program, you'll see that it works the same as the previous *magic8Ball.py* program.

Notice the expression you use as the index into `messages`: `random.randint(0, len(messages) - 1)`. This produces a random number to use for the index, regardless of the size of `messages`. That is, you'll get a random number between `0` and the value of `len(messages) - 1`. The benefit of this approach is that you can easily add and remove strings to the `messages` list without changing other lines of code. If you later update your code, there will be fewer lines you have to change and fewer chances for you to introduce bugs.

List-like Types: Strings and Tuples

Lists aren't the only data types that represent ordered sequences of values. For example, strings and lists are actually similar, if you consider a string to be a "list" of single text characters. Many of the things you can do with lists

can also be done with strings: indexing; slicing; and using them with `for` loops, with `len()`, and with the `in` and `not in` operators. To see this, enter the following into the interactive shell:

```
>>> name = 'Zophie'
>>> name[0]
'Z'
>>> name[-2]
'i'
>>> name[0:4]
'Zoph'
>>> 'Zo' in name
True
>>> 'z' in name
False
>>> 'p' not in name
False
>>> for i in name:
        print('* * * ' + i + ' * * *')


* * * Z * * *
* * * o * * *
* * * p * * *
* * * h * * *
* * * i * * *
* * * e * * *
```

Mutable and Immutable Data Types

But lists and strings are different in an important way. A list value is a *mutable* data type: It can have values added, removed, or changed. However, a string is *immutable*: It cannot be changed. Trying to reassign a single character in a string results in a `TypeError` error, as you can see by entering the following into the interactive shell:

```
>>> name = 'Zophie a cat'
>>> name[7] = 'the'
Traceback (most recent call last):
  File "<pyshell#50>", line 1, in <module>
    name[7] = 'the'
TypeError: 'str' object does not support item assignment
```

The proper way to "mutate" a string is to use slicing and concatenation to build a *new* string by copying from parts of the old string. Enter the following into the interactive shell:

```
>>> name = 'Zophie a cat'
>>> newName = name[0:7] + 'the' + name[8:12]
>>> name
'Zophie a cat'
```

```
>>> newName
'Zophie the cat'
```

We used [0:7] and [8:12] to refer to the characters that we don't wish to replace. Notice that the original 'Zophie a cat' string is not modified because strings are immutable.

Although a list value *is* mutable, the second line in the following code does not modify the list eggs:

```
>>> eggs = [1, 2, 3]
>>> eggs = [4, 5, 6]
>>> eggs
[4, 5, 6]
```

The list value in eggs isn't being changed here; rather, an entirely new and different list value ([4, 5, 6]) is overwriting the old list value ([1, 2, 3]). This is depicted in Figure 4-2.

If you wanted to actually modify the original list in eggs to contain [4, 5, 6], you would have to do something like this:

```
>>> eggs = [1, 2, 3]
>>> del eggs[2]
>>> del eggs[1]
>>> del eggs[0]
>>> eggs.append(4)
>>> eggs.append(5)
>>> eggs.append(6)
>>> eggs
[4, 5, 6]
```

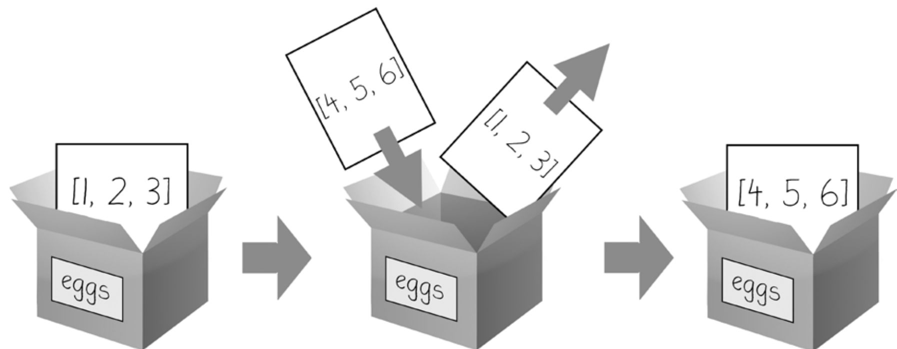


Figure 4-2: When eggs = [4, 5, 6] is executed, the contents of eggs are replaced with a new list value.

In the first example, the list value that eggs ends up with is the same list value it started with. It's just that this list has been changed, rather than overwritten. Figure 4-3 depicts the seven changes made by the first seven lines in the previous interactive shell example.

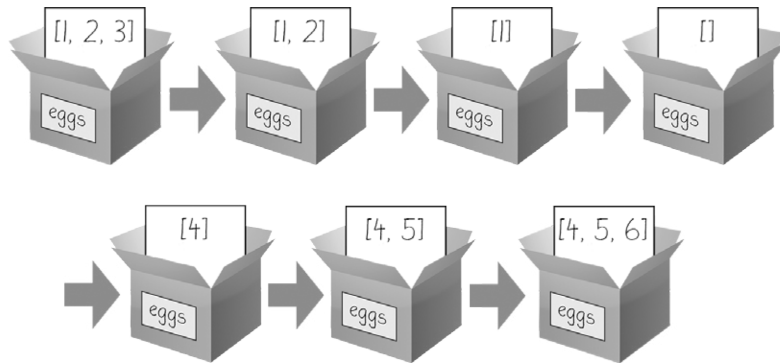


Figure 4-3: The `del` statement and the `append()` method modify the same list value in place.

Changing a value of a mutable data type (like what the `del` statement and `append()` method do in the previous example) changes the value in place, since the variable's value is not replaced with a new list value.

Mutable versus immutable types may seem like a meaningless distinction, but "Passing References" on page 100 will explain the different behavior when calling functions with mutable arguments versus immutable arguments. But first, let's find out about the tuple data type, which is an immutable form of the list data type.

The Tuple Data Type

The *tuple* data type is almost identical to the list data type, except in two ways. First, tuples are typed with parentheses, (and), instead of square brackets, [and]. For example, enter the following into the interactive shell:

```
>>> eggs = ('hello', 42, 0.5)
>>> eggs[0]
'hello'
>>> eggs[1:3]
(42, 0.5)
>>> len(eggs)
3
```

But the main way that tuples are different from lists is that tuples, like strings, are immutable. Tuples cannot have their values modified, appended, or removed. Enter the following into the interactive shell, and look at the `TypeError` error message:

```
>>> eggs = ('hello', 42, 0.5)
>>> eggs[1] = 99
Traceback (most recent call last):
  File "<pyshell#5>", line 1, in <module>
    eggs[1] = 99
TypeError: 'tuple' object does not support item assignment
```

If you have only one value in your tuple, you can indicate this by placing a trailing comma after the value inside the parentheses. Otherwise, Python will think you've just typed a value inside regular parentheses. The comma is what lets Python know this is a tuple value. (Unlike some other programming languages, in Python it's fine to have a trailing comma after the last item in a list or tuple.) Enter the following `type()` function calls into the interactive shell to see the distinction:

```
>>> type(('hello',))
<class 'tuple'>
>>> type(('hello'))
<class 'str'>
```

You can use tuples to convey to anyone reading your code that you don't intend for that sequence of values to change. If you need an ordered sequence of values that never changes, use a tuple. A second benefit of using tuples instead of lists is that, because they are immutable and their contents don't change, Python can implement some optimizations that make code using tuples slightly faster than code using lists.

Converting Types with the list() and tuple() Functions

Just like how `str(42)` will return `'42'`, the string representation of the integer `42`, the functions `list()` and `tuple()` will return list and tuple versions of the values passed to them. Enter the following into the interactive shell, and notice that the return value is of a different data type than the value passed:

```
>>> tuple(['cat', 'dog', 5])
('cat', 'dog', 5)
>>> list(('cat', 'dog', 5))
['cat', 'dog', 5]
>>> list('hello')
['h', 'e', 'l', 'l', 'o']
```

Converting a tuple to a list is handy if you need a mutable version of a tuple value.

References

As you've seen, variables store strings and integer values. Enter the following into the interactive shell:

```
>>> spam = 42
>>> cheese = spam
>>> spam = 100
>>> spam
100
>>> cheese
42
```

You assign 42 to the spam variable, and then you copy the value in spam and assign it to the variable cheese. When you later change the value in spam to 100, this doesn't affect the value in cheese. This is because spam and cheese are different variables that store different values.

But lists don't work this way. When you assign a list to a variable, you are actually assigning a list *reference* to the variable. A reference is a value that points to some bit of data, and a list reference is a value that points to a list. Here is some code that will make this distinction easier to understand. Enter this into the interactive shell:

```
❶ >>> spam = [0, 1, 2, 3, 4, 5]
❷ >>> cheese = spam
❸ >>> cheese[1] = 'Hello!'
   >>> spam
   [0, 'Hello!', 2, 3, 4, 5]
   >>> cheese
   [0, 'Hello!', 2, 3, 4, 5]
```

This might look odd to you. The code changed only the cheese list, but it seems that both the cheese and spam lists have changed.

When you create the list ❶, you assign a reference to it in the spam variable. But the next line ❷ copies only the list reference in spam to cheese, not the list value itself. This means the values stored in spam and cheese now both refer to the same list. There is only one underlying list because the list itself was never actually copied. So when you modify the first element of cheese ❸, you are modifying the same list that spam refers to.

Remember that variables are like boxes that contain values. The previous figures in this chapter show that lists in boxes aren't exactly accurate because list variables don't actually contain lists—they contain *references* to lists. (These references will have ID numbers that Python uses internally, but you can ignore them.) Using boxes as a metaphor for variables, Figure 4-4 shows what happens when a list is assigned to the spam variable.

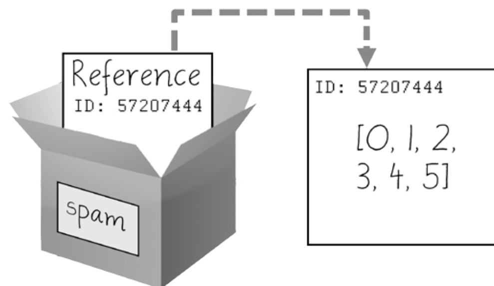


Figure 4-4: spam = [0, 1, 2, 3, 4, 5] stores a reference to a list, not the actual list.

Then, in Figure 4-5, the reference in spam is copied to cheese. Only a new reference was created and stored in cheese, not a new list. Note how both references refer to the same list.

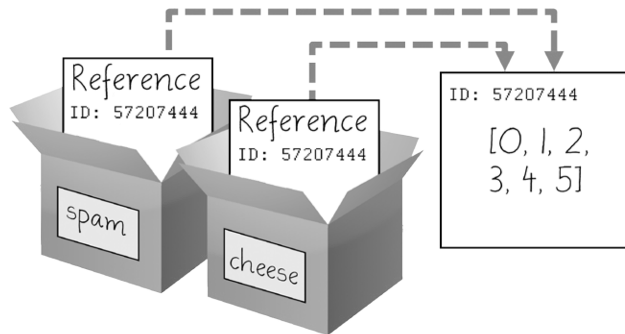


Figure 4-5: spam = cheese *copies the reference, not the list.*

When you alter the list that cheese refers to, the list that spam refers to is also changed, because both cheese and spam refer to the same list. You can see this in Figure 4-6.

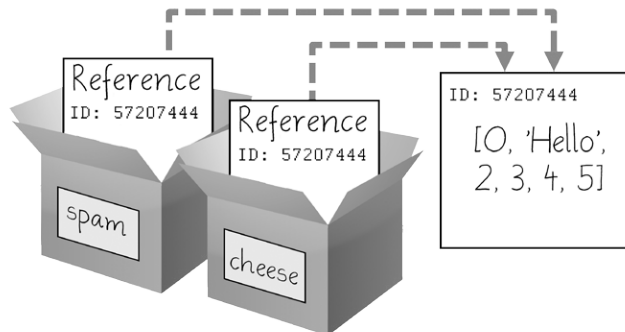


Figure 4-6: cheese[1] = 'Hello!' *modifies the list that both variables refer to.*

Variables will contain references to list values rather than list values themselves. But for strings and integer values, variables simply contain the string or integer value. Python uses references whenever variables must store values of mutable data types, such as lists or dictionaries. For values of immutable data types such as strings, integers, or tuples, Python variables will store the value itself.

Although Python variables technically contain references to list or dictionary values, people often casually say that the variable contains the list or dictionary.

Passing References

References are particularly important for understanding how arguments get passed to functions. When a function is called, the values of the arguments are copied to the parameter variables. For lists (and dictionaries, which I'll describe in the next chapter), this means a copy of the reference is used for the parameter. To see the consequences of this, open a new file editor window, enter the following code, and save it as *passingReference.py*:

```
def eggs(someParameter):
    someParameter.append('Hello')

spam = [1, 2, 3]
eggs(spam)
print(spam)
```

Notice that when `eggs()` is called, a return value is not used to assign a new value to `spam`. Instead, it modifies the list in place, directly. When run, this program produces the following output:

```
[1, 2, 3, 'Hello']
```

Even though `spam` and `someParameter` contain separate references, they both refer to the same list. This is why the `append('Hello')` method call inside the function affects the list even after the function call has returned.

Keep this behavior in mind: Forgetting that Python handles list and dictionary variables this way can lead to confusing bugs.

The copy Module's copy() and deepcopy() Functions

Although passing around references is often the handiest way to deal with lists and dictionaries, if the function modifies the list or dictionary that is passed, you may not want these changes in the original list or dictionary value. For this, Python provides a module named `copy` that provides both the `copy()` and `deepcopy()` functions. The first of these, `copy.copy()`, can be used to make a duplicate copy of a mutable value like a list or dictionary, not just a copy of a reference. Enter the following into the interactive shell:

```
>>> import copy
>>> spam = ['A', 'B', 'C', 'D']
>>> cheese = copy.copy(spam)
>>> cheese[1] = 42
>>> spam
['A', 'B', 'C', 'D']
>>> cheese
['A', 42, 'C', 'D']
```

Now the `spam` and `cheese` variables refer to separate lists, which is why only the list in `cheese` is modified when you assign `42` at index `7`. As you can see in Figure 4-7, the reference ID numbers are no longer the same for both variables because the variables refer to independent lists.

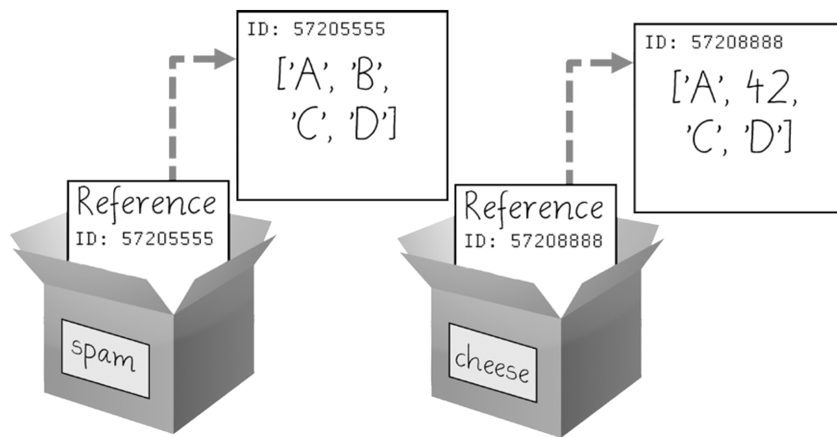


Figure 4-7: `cheese = copy.copy(spam)` creates a second list that can be modified independently of the first.

If the list you need to copy contains lists, then use the `copy.deepcopy()` function instead of `copy.copy()`. The `deepcopy()` function will copy these inner lists as well.

Summary

Lists are useful data types since they allow you to write code that works on a modifiable number of values in a single variable. Later in this book, you will see programs using lists to do things that would be difficult or impossible to do without them.

Lists are mutable, meaning that their contents can change. Tuples and strings, although list-like in some respects, are immutable and cannot be changed. A variable that contains a tuple or string value can be overwritten with a new tuple or string value, but this is not the same thing as modifying the existing value in place—like, say, the `append()` or `remove()` methods do on lists.

Variables do not store list values directly; they store *references* to lists. This is an important distinction when copying variables or passing lists as arguments in function calls. Because the value that is being copied is the list reference, be aware that any changes you make to the list might impact another variable in your program. You can use `copy()` or `deepcopy()` if you want to make changes to a list in one variable without modifying the original list.

Practice Questions

1. What is []?
2. How would you assign the value `'hello'` as the third value in a list stored in a variable named `spam`? (Assume `spam` contains `[2, 4, 6, 8, 10]`.)

For the following three questions, let's say `spam` contains the list `['a', 'b', 'c', 'd']`.

3. What does `spam[int('3' * 2) / 11]` evaluate to?
4. What does `spam[-1]` evaluate to?
5. What does `spam[:2]` evaluate to?

For the following three questions, let's say `bacon` contains the list `[3.14, 'cat', 11, 'cat', True]`.

6. What does `bacon.index('cat')` evaluate to?
7. What does `bacon.append(99)` make the list value in `bacon` look like?
8. What does `bacon.remove('cat')` make the list value in `bacon` look like?
9. What are the operators for list concatenation and list replication?
10. What is the difference between the `append()` and `insert()` list methods?
11. What are two ways to remove values from a list?
12. Name a few ways that list values are similar to string values.
13. What is the difference between lists and tuples?
14. How do you type the tuple value that has just the integer value `42` in it?
15. How can you get the tuple form of a list value? How can you get the list form of a tuple value?
16. Variables that "contain" list values don't actually contain lists directly. What do they contain instead?
17. What is the difference between `copy.copy()` and `copy.deepcopy()`?

Practice Projects

For practice, write programs to do the following tasks.

Comma Code

Say you have a list value like this:

```
spam = ['apples', 'bananas', 'tofu', 'cats']
```

Write a function that takes a list value as an argument and returns a string with all the items separated by a comma and a space, with *and* inserted before the last item. For example, passing the previous `spam` list to the function would return `'apples, bananas, tofu, and cats'`. But your function should be able to work with any list value passed to it.

Character Picture Grid

Say you have a list of lists where each value in the inner lists is a one-character string, like this:

```
grid = [['.', '.', '.', '.', '.', '.'],
        ['.', 'O', 'O', '.', '.', '.'],
        ['O', 'O', 'O', 'O', '.', '.'],
        ['O', 'O', 'O', 'O', 'O', '.'],
        ['.', 'O', 'O', 'O', 'O', 'O'],
        ['O', 'O', 'O', 'O', 'O', '.'],
        ['O', 'O', 'O', 'O', '.', '.'],
        ['.', 'O', 'O', '.', '.', '.'],
        ['.', '.', '.', '.', '.', '.']]
```

You can think of `grid[x][y]` as being the character at the x- and y-coordinates of a "picture" drawn with text characters. The `(0, 0)` origin will be in the upper-left corner, the x-coordinates increase going right, and w the y-coordinates increase going down.

Copy the previous grid value, and write code that uses it to print the image.

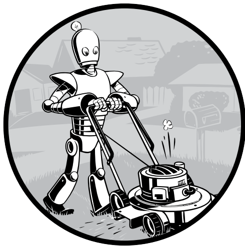
```
..OO.OO..
.OOOOOOO.
.OOOOOOO.
..OOOOO..
...OOO...
....O....
```

Hint: You will need to use a loop in a loop in order to print `grid[0][0]`, then `grid[1][0]`, then `grid[2][0]`, and so on, up to `grid[8][0]`. This will finish the first row, so then print a newline. Then your program should print `grid[0][1]`, then `grid[1][1]`, then `grid[2][1]`, and so on. The last thing your program will print is `grid[8][5]`.

Also, remember to pass the `end` keyword argument to `print()` if you don't want a newline printed automatically after each `print()` call.

5

DICTIONARIES AND STRUCTURING DATA



In this chapter, I will cover the dictionary data type, which provides a flexible way to access and organize data. Then, combining dictionaries with your knowledge of lists from the previous chapter, you'll learn how to create a data structure to model a tic-tac-toe board.

The Dictionary Data Type

Like a list, a *dictionary* is a collection of many values. But unlike indexes for lists, indexes for dictionaries can use many different data types, not just integers. Indexes for dictionaries are called *keys*, and a key with its associated value is called a *key-value pair*.

In code, a dictionary is typed with braces, {}. Enter the following into the interactive shell:

```
>>> myCat = {'size': 'fat', 'color': 'gray', 'disposition': 'loud'}
```

This assigns a dictionary to the `myCat` variable. This dictionary's keys are `'size'`, `'color'`, and `'disposition'`. The values for these keys are `'fat'`, `'gray'`, and `'loud'`, respectively. You can access these values through their keys:

```
>>> myCat['size']
'fat'
>>> 'My cat has ' + myCat['color'] + ' fur.'
'My cat has gray fur.'
```

Dictionaries can still use integer values as keys, just like lists use integers for indexes, but they do not have to start at `0` and can be any number.

```
>>> spam = {12345: 'Luggage Combination', 42: 'The Answer'}
```

Dictionaries vs. Lists

Unlike lists, items in dictionaries are unordered. The first item in a list named `spam` would be `spam[0]`. But there is no "first" item in a dictionary. While the order of items matters for determining whether two lists are the same, it does not matter in what order the key-value pairs are typed in a dictionary. Enter the following into the interactive shell:

```
>>> spam = ['cats', 'dogs', 'moose']
>>> bacon = ['dogs', 'moose', 'cats']
>>> spam == bacon
False
>>> eggs = {'name': 'Zophie', 'species': 'cat', 'age': '8'}
>>> ham = {'species': 'cat', 'age': '8', 'name': 'Zophie'}
>>> eggs == ham
True
```

Because dictionaries are not ordered, they can't be sliced like lists.

Trying to access a key that does not exist in a dictionary will result in a `KeyError` error message, much like a list's "out-of-range" `IndexError` error message. Enter the following into the interactive shell, and notice the error message that shows up because there is no `'color'` key:

```
>>> spam = {'name': 'Zophie', 'age': 7}
>>> spam['color']
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    spam['color']
KeyError: 'color'
```

Though dictionaries are not ordered, the fact that you can have arbitrary values for the keys allows you to organize your data in powerful ways. Say you wanted your program to store data about your friends' birthdays. You can use a dictionary with the names as keys and the birthdays as values. Open a new file editor window and enter the following code. Save it as *birthdays.py*.

```
❶ birthdays = {'Alice': 'Apr 1', 'Bob': 'Dec 12', 'Carol': 'Mar 4'}

  while True:
      print('Enter a name: (blank to quit)')
      name = input()
      if name == '':
          break

❷     if name in birthdays:
❸         print(birthdays[name] + ' is the birthday of ' + name)
      else:
          print('I do not have birthday information for ' + name)
          print('What is their birthday?')
          bday = input()
❹         birthdays[name] = bday
          print('Birthday database updated.')
```

You create an initial dictionary and store it in `birthdays` ❶. You can see if the entered name exists as a key in the dictionary with the `in` keyword ❷, just as you did for lists. If the name is in the dictionary, you access the associated value using square brackets ❸; if not, you can add it using the same square bracket syntax combined with the assignment operator ❹.

When you run this program, it will look like this:

```
Enter a name: (blank to quit)
Alice
Apr 1 is the birthday of Alice
Enter a name: (blank to quit)
Eve
I do not have birthday information for Eve
What is their birthday?
Dec 5
Birthday database updated.
Enter a name: (blank to quit)
Eve
Dec 5 is the birthday of Eve
Enter a name: (blank to quit)
```

Of course, all the data you enter in this program is forgotten when the program terminates. You'll learn how to save data to files on the hard drive in Chapter 8.

The keys(), values(), and items() Methods

There are three dictionary methods that will return list-like values of the dictionary's keys, values, or both keys and values: `keys()`, `values()`, and `items()`. The values returned by these methods are not true lists: They cannot be modified and do not have an `append()` method. But these data types (`dict_keys`,

dict_values, and dict_items, respectively) *can* be used in for loops. To see how these methods work, enter the following into the interactive shell:

```
>>> spam = {'color': 'red', 'age': 42}
>>> for v in spam.values():
        print(v)

red
42
```

Here, a for loop iterates over each of the values in the spam dictionary. A for loop can also iterate over the keys or both keys and values:

```
>>> for k in spam.keys():
        print(k)

color
age
>>> for i in spam.items():
        print(i)

('color', 'red')
('age', 42)
```

Using the keys(), values(), and items() methods, a for loop can iterate over the keys, values, or key-value pairs in a dictionary, respectively. Notice that the values in the dict_items value returned by the items() method are tuples of the key and value.

If you want a true list from one of these methods, pass its list-like return value to the list() function. Enter the following into the interactive shell:

```
>>> spam = {'color': 'red', 'age': 42}
>>> spam.keys()
dict_keys(['color', 'age'])
>>> list(spam.keys())
['color', 'age']
```

The list(spam.keys()) line takes the dict_keys value returned from keys() and passes it to list(), which then returns a list value of ['color', 'age'].

You can also use the multiple assignment trick in a for loop to assign the key and value to separate variables. Enter the following into the interactive shell:

```
>>> spam = {'color': 'red', 'age': 42}
>>> for k, v in spam.items():
        print('Key: ' + k + ' Value: ' + str(v))

Key: age Value: 42
Key: color Value: red
```

Checking Whether a Key or Value Exists in a Dictionary

Recall from the previous chapter that the in and not in operators can check whether a value exists in a list. You can also use these operators to see whether a certain key or value exists in a dictionary. Enter the following into the interactive shell:

```
>>> spam = {'name': 'Zophie', 'age': 7}
>>> 'name' in spam.keys()
True
>>> 'Zophie' in spam.values()
True
>>> 'color' in spam.keys()
False
>>> 'color' not in spam.keys()
True
>>> 'color' in spam
False
```

In the previous example, notice that 'color' in spam is essentially a shorter version of writing 'color' in spam.keys(). This is always the case: If you ever want to check whether a value is (or isn't) a key in the dictionary, you can simply use the in (or not in) keyword with the dictionary value itself.

The get() Method

It's tedious to check whether a key exists in a dictionary before accessing that key's value. Fortunately, dictionaries have a get() method that takes two arguments: the key of the value to retrieve and a fallback value to return if that key does not exist.

Enter the following into the interactive shell:

```
>>> picnicItems = {'apples': 5, 'cups': 2}
>>> 'I am bringing ' + str(picnicItems.get('cups', 0)) + ' cups.'
'I am bringing 2 cups.'
>>> 'I am bringing ' + str(picnicItems.get('eggs', 0)) + ' eggs.'
'I am bringing 0 eggs.'
```

Because there is no 'eggs' key in the picnicItems dictionary, the default value 0 is returned by the get() method. Without using get(), the code would have caused an error message, such as in the following example:

```
>>> picnicItems = {'apples': 5, 'cups': 2}
>>> 'I am bringing ' + str(picnicItems['eggs']) + ' eggs.'
Traceback (most recent call last):
  File "<pyshell#34>", line 1, in <module>
    'I am bringing ' + str(picnicItems['eggs']) + ' eggs.'
KeyError: 'eggs'
```

The setdefault() Method

You'll often have to set a value in a dictionary for a certain key only if that key does not already have a value. The code looks something like this:

```
spam = {'name': 'Pooka', 'age': 5}
if 'color' not in spam:
    spam['color'] = 'black'
```

The `setdefault()` method offers a way to do this in one line of code. The first argument passed to the method is the key to check for, and the second argument is the value to set at that key if the key does not exist. If the key does exist, the `setdefault()` method returns the key's value. Enter the following into the interactive shell:

```
>>> spam = {'name': 'Pooka', 'age': 5}
>>> spam.setdefault('color', 'black')
'black'
>>> spam
{'color': 'black', 'age': 5, 'name': 'Pooka'}
>>> spam.setdefault('color', 'white')
'black'
>>> spam
{'color': 'black', 'age': 5, 'name': 'Pooka'}
```

The first time `setdefault()` is called, the dictionary in `spam` changes to `{'color': 'black', 'age': 5, 'name': 'Pooka'}`. The method returns the value `'black'` because this is now the value set for the key `'color'`. When `spam.setdefault('color', 'white')` is called next, the value for that key is *not* changed to `'white'` because `spam` already has a key named `'color'`.

The `setdefault()` method is a nice shortcut to ensure that a key exists. Here is a short program that counts the number of occurrences of each letter in a string. Open the file editor window and enter the following code, saving it as *characterCount.py*:

```
message = 'It was a bright cold day in April, and the clocks were striking thirteen.'
count = {}

for character in message:
    count.setdefault(character, 0)
    count[character] = count[character] + 1

print(count)
```

The program loops over each character in the `message` variable's string, counting how often each character appears. The `setdefault()` method call ensures that the key is in the `count` dictionary (with a default value of `0`)

so the program doesn't throw a `KeyError` error when `count[character] = count[character] + 1` is executed. When you run this program, the output will look like this:

```
{' ': 13, ',': 1, '.': 1, 'A': 1, 'I': 1, 'a': 4, 'c': 3, 'b': 1, 'e': 5, 'd': 3, 'g': 2, 'i':
6, 'h': 3, 'k': 2, 'l': 3, 'o': 2, 'n': 4, 'p': 1, 's': 3, 'r': 5, 't': 6, 'w': 2, 'y': 1}
```

From the output, you can see that the lowercase letter *c* appears 3 times, the space character appears 13 times, and the uppercase letter *A* appears 1 time. This program will work no matter what string is inside the `message` variable, even if the string is millions of characters long!

Pretty Printing

If you import the `pprint` module into your programs, you'll have access to the `pprint()` and `pformat()` functions that will "pretty print" a dictionary's values. This is helpful when you want a cleaner display of the items in a dictionary than what `print()` provides. Modify the previous *characterCount.py* program and save it as *prettyCharacterCount.py*.

```
import pprint
message = 'It was a bright cold day in April, and the clocks were striking
thirteen.'
count = {}

for character in message:
    count.setdefault(character, 0)
    count[character] = count[character] + 1

pprint.pprint(count)
```

This time, when the program is run, the output looks much cleaner, with the keys sorted.

```
{' ': 13,
 ',': 1,
 '.': 1,
 'A': 1,
 'I': 1,
 'a': 4,
 'b': 1,
 'c': 3,
 'd': 3,
 'e': 5,
 'g': 2,
 'h': 3,
 'i': 6,
```

```
 'k': 2,
 'l': 3,
 'n': 4,
 'o': 2,
 'p': 1,
 'r': 5,
 's': 3,
 't': 6,
 'w': 2,
 'y': 1}
```

The `pprint.pprint()` function is especially helpful when the dictionary itself contains nested lists or dictionaries.

If you want to obtain the prettified text as a string value instead of displaying it on the screen, call `pprint.pformat()` instead. These two lines are equivalent to each other:

```
pprint.pprint(someDictionaryValue)
print(pprint.pformat(someDictionaryValue))
```

Using Data Structures to Model Real-World Things

Even before the Internet, it was possible to play a game of chess with someone on the other side of the world. Each player would set up a chessboard at their home and then take turns mailing a postcard to each other describing each move. To do this, the players needed a way to unambiguously describe the state of the board and their moves.

In *algebraic chess notation*, the spaces on the chessboard are identified by a number and letter coordinate, as in Figure 5-1.

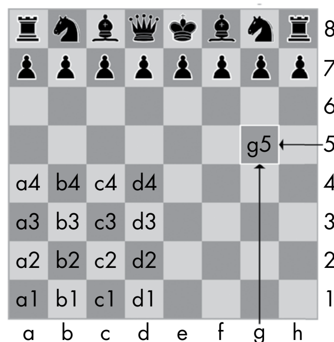


Figure 5-1: The coordinates of a chessboard in algebraic chess notation

The chess pieces are identified by letters: *K* for king, *Q* for queen, *R* for rook, *B* for bishop, and *N* for knight. Describing a move uses the letter of the piece and the coordinates of its destination. A pair of these moves describes

what happens in a single turn (with white going first); for instance, the notation *2. Nf3 Nc6* indicates that white moved a knight to f3 and black moved a knight to c6 on the second turn of the game.

There's a bit more to algebraic notation than this, but the point is that you can use it to unambiguously describe a game of chess without needing to be in front of a chessboard. Your opponent can even be on the other side of the world! In fact, you don't even need a physical chess set if you have a good memory: You can just read the mailed chess moves and update boards you have in your imagination.

Computers have good memories. A program on a modern computer can easily store billions of strings like `'2. Nf3 Nc6'`. This is how computers can play chess without having a physical chessboard. They model data to represent a chessboard, and you can write code to work with this model.

This is where lists and dictionaries can come in. You can use them to model real-world things, like chessboards. For the first example, you'll use a game that's a little simpler than chess: tic-tac-toe.

A Tic-Tac-Toe Board

A tic-tac-toe board looks like a large hash symbol (#) with nine slots that can each contain an *X*, an *O*, or a blank. To represent the board with a dictionary, you can assign each slot a string-value key, as shown in Figure 5-2.

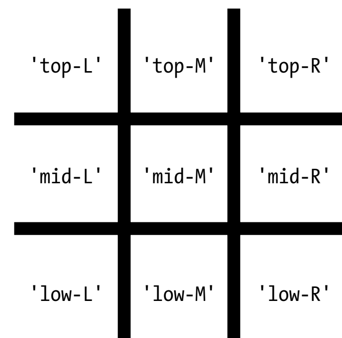


Figure 5-2: The slots of a tic-tac-toe board with their corresponding keys

You can use string values to represent what's in each slot on the board: `'X'`, `'O'`, or `' '` (a space character). Thus, you'll need to store nine strings. You can use a dictionary of values for this. The string value with the key `'top-R'` can represent the top-right corner, the string value with the key `'low-L'` can represent the bottom-left corner, the string value with the key `'mid-M'` can represent the middle, and so on.

This dictionary is a data structure that represents a tic-tac-toe board. Store this board-as-a-dictionary in a variable named `theBoard`. Open a new file editor window, and enter the following source code, saving it as *ticTacToe.py*:

```
theBoard = {'top-L': ' ', 'top-M': ' ', 'top-R': ' ',
            'mid-L': ' ', 'mid-M': ' ', 'mid-R': ' ',
            'low-L': ' ', 'low-M': ' ', 'low-R': ' '}
```

The data structure stored in the `theBoard` variable represents the tic-tac-toe board in Figure 5-3.

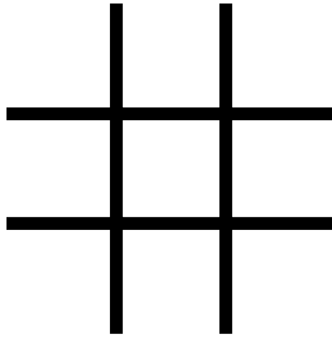


Figure 5-3: An empty tic-tac-toe board

Since the value for every key in `theBoard` is a single-space string, this dictionary represents a completely clear board. If player X went first and chose the middle space, you could represent that board with this dictionary:

```
theBoard = {'top-L': ' ', 'top-M': ' ', 'top-R': ' ',
            'mid-L': ' ', 'mid-M': 'X', 'mid-R': ' ',
            'low-L': ' ', 'low-M': ' ', 'low-R': ' '}
```

The data structure in `theBoard` now represents the tic-tac-toe board in Figure 5-4.

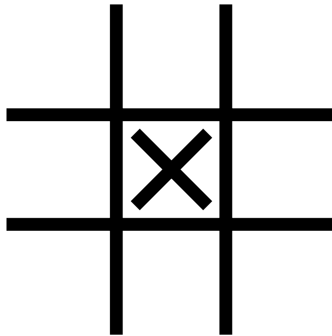


Figure 5-4: The first move

A board where player O has won by placing *O*s across the top might look like this:

```
theBoard = {'top-L': 'O', 'top-M': 'O', 'top-R': 'O',
            'mid-L': 'X', 'mid-M': 'X', 'mid-R': ' ',
            'low-L': ' ', 'low-M': ' ', 'low-R': 'X'}
```

The data structure in `theBoard` now represents the tic-tac-toe board in Figure 5-5.

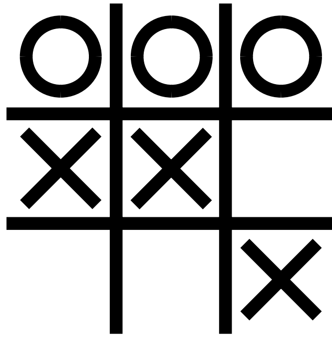


Figure 5-5: Player O wins.

Of course, the player sees only what is printed to the screen, not the contents of variables. Let's create a function to print the board dictionary onto the screen. Make the following addition to *ticTacToe.py* (new code is in bold):

```
theBoard = {'top-L': ' ', 'top-M': ' ', 'top-R': ' ',
            'mid-L': ' ', 'mid-M': ' ', 'mid-R': ' ',
            'low-L': ' ', 'low-M': ' ', 'low-R': ' '}
def printBoard(board):
    print(board['top-L'] + '|' + board['top-M'] + '|' + board['top-R'])
    print('-+-+-')
    print(board['mid-L'] + '|' + board['mid-M'] + '|' + board['mid-R'])
    print('-+-+-')
    print(board['low-L'] + '|' + board['low-M'] + '|' + board['low-R'])
printBoard(theBoard)
```

When you run this program, `printBoard()` will print out a blank tic-tac-toe board.

```
 | |
-+-+-
 | |
-+-+-
 | |
```

The `printBoard()` function can handle any tic-tac-toe data structure you pass it. Try changing the code to the following:

```
theBoard = {'top-L': 'O', 'top-M': 'O', 'top-R': 'O', 'mid-L': 'X', 'mid-M':
'X', 'mid-R': ' ', 'low-L': ' ', 'low-M': ' ', 'low-R': 'X'}

def printBoard(board):
    print(board['top-L'] + '|' + board['top-M'] + '|' + board['top-R'])
    print('-+-+-')
    print(board['mid-L'] + '|' + board['mid-M'] + '|' + board['mid-R'])
    print('-+-+-')
    print(board['low-L'] + '|' + board['low-M'] + '|' + board['low-R'])
printBoard(theBoard)
```

Now when you run this program, the new board will be printed to the screen.

```
O|O|O
-+-+-
X|X|
-+-+-
 | |X
```

Because you created a data structure to represent a tic-tac-toe board and wrote code in `printBoard()` to interpret that data structure, you now have a program that "models" the tic-tac-toe board. You could have organized your data structure differently (for example, using keys like `'TOP-LEFT'` instead of `'top-L'`), but as long as the code works with your data structures, you will have a correctly working program.

For example, the `printBoard()` function expects the tic-tac-toe data structure to be a dictionary with keys for all nine slots. If the dictionary you passed was missing, say, the `'mid-L'` key, your program would no longer work.

```
O|O|O
-+-+-
Traceback (most recent call last):
  File "ticTacToe.py", line 10, in <module>
    printBoard(theBoard)
  File "ticTacToe.py", line 6, in printBoard
    print(board['mid-L'] + '|' + board['mid-M'] + '|' + board['mid-R'])
KeyError: 'mid-L'
```

Now let's add code that allows the players to enter their moves. Modify the *ticTacToe.py* program to look like this:

```
theBoard = {'top-L': ' ', 'top-M': ' ', 'top-R': ' ', 'mid-L': ' ', 'mid-M': '
', 'mid-R': ' ', 'low-L': ' ', 'low-M': ' ', 'low-R': ' '}

def printBoard(board):
    print(board['top-L'] + '|' + board['top-M'] + '|' + board['top-R'])
    print('-+-+-')
    print(board['mid-L'] + '|' + board['mid-M'] + '|' + board['mid-R'])
    print('-+-+-')
    print(board['low-L'] + '|' + board['low-M'] + '|' + board['low-R'])
turn = 'X'
for i in range(9):
❶   printBoard(theBoard)
    print('Turn for ' + turn + '. Move on which space?')
❷   move = input()
❸   theBoard[move] = turn
❹   if turn == 'X':
        turn = 'O'
    else:
        turn = 'X'
printBoard(theBoard)
```

The new code prints out the board at the start of each new turn ❶, gets the active player's move ❷, updates the game board accordingly ❸, and then swaps the active player ❹ before moving on to the next turn.

When you run this program, it will look something like this:

```
 | |
-+-+-
 | |
-+-+-
 | |
Turn for X. Move on which space?
mid-M
 | |
-+-+-
 |X|
-+-+-
 | |
Turn for O. Move on which space?
low-L
 | |
-+-+-
 |X|
-+-+-
O| |

--snip--

O|O|X
-+-+-
X|X|O
-+-+-
O| |X
Turn for X. Move on which space?
low-M
O|O|X
-+-+-
X|X|O
-+-+-
O|X|X
```

This isn't a complete tic-tac-toe game—for instance, it doesn't ever check whether a player has won—but it's enough to see how data structures can be used in programs.

NOTE *If you are curious, the source code for a complete tic-tac-toe program is described in the resources available from* http://nostarch.com/automatestuff/.

Nested Dictionaries and Lists

Modeling a tic-tac-toe board was fairly simple: The board needed only a single dictionary value with nine key-value pairs. As you model more complicated things, you may find you need dictionaries and lists that contain

other dictionaries and lists. Lists are useful to contain an ordered series of values, and dictionaries are useful for associating keys with values. For example, here's a program that uses a dictionary that contains other dictionaries in order to see who is bringing what to a picnic. The `totalBrought()` function can read this data structure and calculate the total number of an item being brought by all the guests.

```
allGuests = {'Alice': {'apples': 5, 'pretzels': 12},
             'Bob': {'ham sandwiches': 3, 'apples': 2},
             'Carol': {'cups': 3, 'apple pies': 1}}

def totalBrought(guests, item):
    numBrought = 0
❶   for k, v in guests.items():
❷       numBrought = numBrought + v.get(item, 0)
    return numBrought

print('Number of things being brought:')
print(' - Apples         ' + str(totalBrought(allGuests, 'apples')))
print(' - Cups           ' + str(totalBrought(allGuests, 'cups')))
print(' - Cakes          ' + str(totalBrought(allGuests, 'cakes')))
print(' - Ham Sandwiches ' + str(totalBrought(allGuests, 'ham sandwiches')))
print(' - Apple Pies     ' + str(totalBrought(allGuests, 'apple pies')))
```

Inside the `totalBrought()` function, the `for` loop iterates over the key-value pairs in `guests` ❶. Inside the loop, the string of the guest's name is assigned to `k`, and the dictionary of picnic items they're bringing is assigned to `v`. If the item parameter exists as a key in this dictionary, it's value (the quantity) is added to `numBrought` ❷. If it does not exist as a key, the `get()` method returns `0` to be added to `numBrought`.

The output of this program looks like this:

```
Number of things being brought:
- Apples 7
- Cups 3
- Cakes 0
- Ham Sandwiches 3
- Apple Pies     1
```

This may seem like such a simple thing to model that you wouldn't need to bother with writing a program to do it. But realize that this same `totalBrought()` function could easily handle a dictionary that contains thousands of guests, each bringing *thousands* of different picnic items. Then having this information in a data structure along with the `totalBrought()` function would save you a lot of time!

You can model things with data structures in whatever way you like, as long as the rest of the code in your program can work with the data model correctly. When you first begin programming, don't worry so much about

the “right” way to model data. As you gain more experience, you may come up with more efficient models, but the important thing is that the data model works for your program’s needs.

Summary

You learned all about dictionaries in this chapter. Lists and dictionaries are values that can contain multiple values, including other lists and dictionaries. Dictionaries are useful because you can map one item (the key) to another (the value), as opposed to lists, which simply contain a series of values in order. Values inside a dictionary are accessed using square brackets just as with lists. Instead of an integer index, dictionaries can have keys of a variety of data types: integers, floats, strings, or tuples. By organizing a program’s values into data structures, you can create representations of real-world objects. You saw an example of this with a tic-tac-toe board.

That just about covers all the basic concepts of Python programming! You’ll continue to learn new concepts throughout the rest of this book, but you now know enough to start writing some useful programs that can automate tasks. You might not think you have enough Python knowledge to do things such as download web pages, update spreadsheets, or send text messages, but that’s where Python modules come in! These modules, written by other programmers, provide functions that make it easy for you to do all these things. So let’s learn how to write real programs to do useful automated tasks.

Practice Questions

1. What does the code for an empty dictionary look like?
2. What does a dictionary value with a key `'foo'` and a value `42` look like?
3. What is the main difference between a dictionary and a list?
4. What happens if you try to access `spam['foo']` if `spam` is `{'bar': 100}`?
5. If a dictionary is stored in `spam`, what is the difference between the expressions `'cat' in spam` and `'cat' in spam.keys()`?
6. If a dictionary is stored in `spam`, what is the difference between the expressions `'cat' in spam` and `'cat' in spam.values()`?
7. What is a shortcut for the following code?

```
if 'color' not in spam:
    spam['color'] = 'black'
```

8. What module and function can be used to “pretty print” dictionary values?

Practice Projects

For practice, write programs to do the following tasks.

Fantasy Game Inventory

You are creating a fantasy video game. The data structure to model the player's inventory will be a dictionary where the keys are string values describing the item in the inventory and the value is an integer value detailing how many of that item the player has. For example, the dictionary value `{'rope': 1, 'torch': 6, 'gold coin': 42, 'dagger': 1, 'arrow': 12}` means the player has 1 rope, 6 torches, 42 gold coins, and so on.

Write a function named `displayInventory()` that would take any possible "inventory" and display it like the following:

```
Inventory:
12 arrow
42 gold coin
1 rope
6 torch
1 dagger
Total number of items: 62
```

Hint: You can use a `for` loop to loop through all the keys in a dictionary.

```
# inventory.py
stuff = {'rope': 1, 'torch': 6, 'gold coin': 42, 'dagger': 1, 'arrow': 12}

def displayInventory(inventory):
    print("Inventory:")
    item_total = 0
    for k, v in inventory.items():
        print(str(v) + ' ' + k)
        item_total += v
    print("Total number of items: " + str(item_total))

displayInventory(stuff)
```

List to Dictionary Function for Fantasy Game Inventory

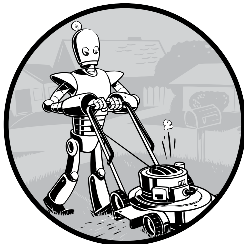
Imagine that a vanquished dragon's loot is represented as a list of strings like this:

```
dragonLoot = ['gold coin', 'dagger', 'gold coin', 'gold coin', 'ruby']
```

Write a function named `addToInventory(inventory, addedItems)`, where the `inventory` parameter is a dictionary representing the player's inventory (like in the previous project) and the `addedItems` parameter is a list like `dragonLoot`.

A

INSTALLING THIRD-PARTY MODULES



Beyond the standard library of modules packaged with Python, other developers have written their own modules to extend Python's capabilities even further. The primary way to install third-party modules is to use Python's pip tool. This tool securely downloads and installs Python modules onto your computer from *https://pypi.python.org/*, the website of the Python Software Foundation. PyPI, or the Python Package Index, is a sort of free app store for Python modules.

The pip Tool

The executable file for the pip tool is called *pip* on Windows and *pip3* on OS X and Linux. On Windows, you can find pip at *C:\Python34\Scripts\pip.exe*. On OS X, it is in */Library/Frameworks/Python.framework/Versions/3.4/bin/pip3*. On Linux, it is in */usr/bin/pip3*.

While pip comes automatically installed with Python 3.4 on Windows and OS X, you must install it separately on Linux. To install pip3 on Ubuntu or Debian Linux, open a new Terminal window and enter **`sudo apt-get install python3-pip`**. To install pip3 on Fedora Linux, enter **`sudo yum install python3 -pip`** into a Terminal window. You will need to enter the administrator password for your computer in order to install this software.

Installing Third-Party Modules

The pip tool is meant to be run from the command line: You pass it the command `install` followed by the name of the module you want to install. For example, on Windows you would enter `pip install` *`ModuleName`*, where *`ModuleName`* is the name of the module. On OS X and Linux, you'll have to run pip3 with the `sudo` prefix to grant administrative privileges to install the module. You would need to type `sudo pip3 install` *`ModuleName`*.

If you already have the module installed but would like to upgrade it to the latest version available on PyPI, run `pip install -U` *`ModuleName`* (or `pip3 install -U` *`ModuleName`* on OS X and Linux).

After installing the module, you can test that it installed successfully by running `import` *`ModuleName`* in the interactive shell. If no error messages are displayed, you can assume the module was installed successfully.

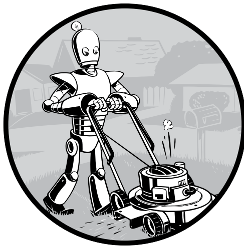
You can install all of the modules covered in this book by running the commands listed next. (Remember to replace `pip` with `pip3` if you're on OS X or Linux.)

- `pip install send2trash`
- `pip install requests`
- `pip install beautifulsoup4`
- `pip install selenium`
- `pip install openpyxl`
- `pip install PyPDF2`
- `pip install python-docx` (install `python-docx`, not `docx`)
- `pip install imapclient`
- `pip install pyzmail`
- `pip install twilio`
- `pip install pillow`
- `pip install pyobjc-core` (on OS X only)
- `pip install pyobjc` (on OS X only)
- `pip install python3-xlib` (on Linux only)
- `pip install pyautogui`

NOTE *For OS X users: The `pyobjc` module can take 20 minutes or longer to install, so don't be alarmed if it takes a while. You should also install the `pyobjc-core` module first, which will reduce the overall installation time.*

B

RUNNING PROGRAMS



If you have a program open in IDLE's file editor, running it is a simple matter of pressing F5 or selecting the Run ▸ Run Module menu item. This is an easy way to run programs while writing them, but opening IDLE to run your finished programs can be a burden. There are more convenient ways to execute Python scripts.

Shebang Line

The first line of all your Python programs should be a *shebang* line, which tells your computer that you want Python to execute this program. The shebang line begins with `#!`, but the rest depends on your operating system.

- On Windows, the shebang line is `#! python3`.
- On OS X, the shebang line is `#! /usr/bin/env python3`.
- On Linux, the shebang line is `#! /usr/bin/python3`.

You will be able to run Python scripts from IDLE without the shebang line, but the line is needed to run them from the command line.

Running Python Programs on Windows

On Windows, the Python 3.4 interpreter is located at *C:\Python34\python.exe*. Alternatively, the convenient *py.exe* program will read the shebang line at the top of the *.py* file's source code and run the appropriate version of Python for that script. The *py.exe* program will make sure to run the Python program with the correct version of Python if multiple versions are installed on your computer.

To make it convenient to run your Python program, create a *.bat batch file* for running the Python program with *py.exe*. To make a batch file, make a new text file containing a single line like the following:

```
@py.exe C:\path\to\your\pythonScript.py %*
```

Replace this path with the absolute path to your own program, and save this file with a *.bat* file extension (for example, *pythonScript.bat*). This batch file will keep you from having to type the full absolute path for the Python program every time you want to run it. I recommend you place all your batch and *.py* files in a single folder, such as *C:\MyPythonScripts* or *C:\Users\YourName\PythonScripts*.

The *C:\MyPythonScripts* folder should be added to the system path on Windows so that you can run the batch files in it from the Run dialog. To do this, modify the `PATH` environment variable. Click the **Start** button and type **Edit environment variables for your account**. This option should auto-complete after you've begun to type it. The Environment Variables window that appears will look like Figure B-1.

From System variables, select the Path variable and click **Edit**. In the Value text field, append a semicolon, type **C:\MyPythonScripts**, and then click **OK**. Now you can run any Python script in the *C:\MyPythonScripts* folder by simply pressing WIN-R and entering the script's name. Running `pythonScript`, for instance, will run *pythonScript.bat*, which in turn will save you from having to run the whole command `py.exe C:\MyPythonScripts\pythonScript.py` from the Run dialog.

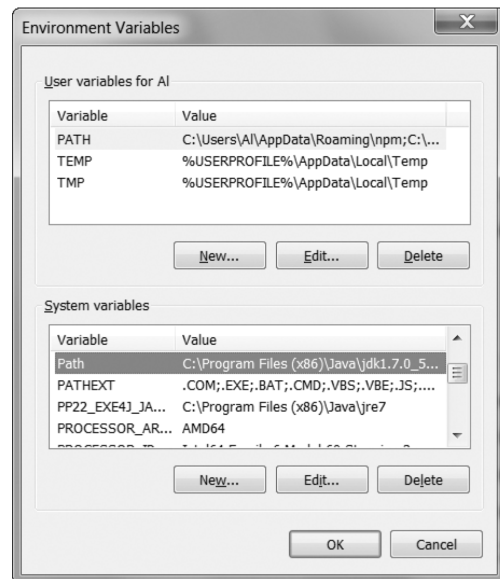


Figure B-1: The Environment Variables window on Windows

Running Python Programs on OS X and Linux

On OS X, selecting Applications ▸ Utilities ▸ Terminal will bring up a *Terminal* window. A Terminal window is a way to enter commands on your computer using only text, rather than clicking through a graphic interface. To bring up the Terminal window on Ubuntu Linux, press the WIN (or SUPER) key to bring up Dash and type in **Terminal**.

The Terminal window will begin in the home folder of your user account. If my username is *asweigart*, the home folder will be */Users/asweigart* on OS X and */home/asweigart* on Linux. The tilde (~) character is a shortcut for your home folder, so you can enter `cd ~` to change to your home folder. You can also use the `cd` command to change the current working directory to any other directory. On both OS X and Linux, the `pwd` command will print the current working directory.

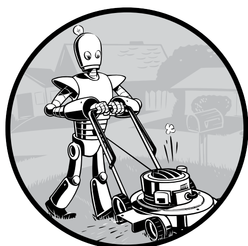
To run your Python programs, save your *.py* file to your home folder. Then, change the *.py* file's permissions to make it executable by running `chmod +x pythonScript.py`. File permissions are beyond the scope of this book, but you will need to run this command on your Python file if you want to run the program from the Terminal window. Once you do so, you will be able to run your script whenever you want by opening a Terminal window and entering `./pythonScript.py`. The shebang line at the top of the script will tell the operating system where to locate the Python interpreter.

Running Python Programs with Assertions Disabled

You can disable the `assert` statements in your Python programs for a slight performance improvement. When running Python from the terminal, include the `-O` switch after `python` or `python3` and before the name of the *.py* file. This will run an optimized version of your program that skips the assertion checks.

C

ANSWERS TO THE PRACTICE QUESTIONS



This appendix contains the answers to the practice problems at the end of each chapter. I highly recommend that you take the time to work through these problems. Programming is more than memorizing syntax and a list of function names. As when learning a foreign language, the more practice you put into it, the more you will get out of it. There are many websites with practice programming problems as well. You can find a list of these at *http://nostarch.com/automatestuff/*.

Chapter 1

1. The operators are `+`, `-`, `*`, and `/`. The values are `'hello'`, `-88.8`, and `5`.
2. The string is `'spam'`; the variable is `spam`. Strings always start and end with quotes.
3. The three data types introduced in this chapter are integers, floating-point numbers, and strings.
4. An expression is a combination of values and operators. All expressions evaluate (that is, reduce) to a single value.
5. An expression evaluates to a single value. A statement does not.
6. The `bacon` variable is set to `20`. The `bacon + 1` expression does not reassign the value in `bacon` (that would need an assignment statement: `bacon = bacon + 1`).
7. Both expressions evaluate to the string `'spamspamspam'`.
8. Variable names cannot begin with a number.
9. The `int()`, `float()`, and `str()` functions will evaluate to the integer, floating-point number, and string versions of the value passed to them.
10. The expression causes an error because `99` is an integer, and only strings can be concatenated to other strings with the `+` operator. The correct way is `I have eaten ' + str(99) + ' burritos.'`.

Chapter 2

1. `True` and `False`, using capital *T* and *F*, with the rest of the word in lowercase
2. `and`, `or`, and `not`
3. `True and True` is `True`.
   `True and False` is `False`.
   `False and True` is `False`.
   `False and False` is `False`.
   `True or True` is `True`.
   `True or False` is `True`.
   `False or True` is `True`.
   `False or False` is `False`.
   `not True` is `False`.
   `not False` is `True`.
4. ```
   False
   False
   True
   False
   False
   True
   ```
   ```

5. `==`, `!=`, `<`, `>`, `<=`, and `>=`.
6. `==` is the equal to operator that compares two values and evaluates to a Boolean, while `=` is the assignment operator that stores a value in a variable.
7. A condition is an expression used in a flow control statement that evaluates to a Boolean value.
8. The three blocks are everything inside the `if` statement and the lines `print('bacon')` and `print('ham')`.

```
print('eggs')
if spam > 5:
    print('bacon')
else:
    print('ham')
print('spam')
```

9. The code:

```
if spam == 1:
    print('Hello')
elif spam == 2:
    print('Howdy')
else:
    print('Greetings!')
```

10. Press CTRL-C to stop a program stuck in an infinite loop.
11. The `break` statement will move the execution outside and just after a loop. The `continue` statement will move the execution to the start of the loop.
12. They all do the same thing. The `range(10)` call ranges from `0` up to (but not including) `10`, `range(0, 10)` explicitly tells the loop to start at `0`, and `range(0, 10, 1)` explicitly tells the loop to increase the variable by `1` on each iteration.
13. The code:

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

and:

```
i = 1
while i <= 10:
    print(i)
    i = i + 1
```

14. This function can be called with `spam.bacon()`.

Chapter 3

1. Functions reduce the need for duplicate code. This makes programs shorter, easier to read, and easier to update.
2. The code in a function executes when the function is called, not when the function is defined.
3. The `def` statement defines (that is, creates) a function.
4. A function consists of the `def` statement and the code in its `def` clause. A function call is what moves the program execution into the function, and the function call evaluates to the function's return value.
5. There is one global scope, and a local scope is created whenever a function is called.
6. When a function returns, the local scope is destroyed, and all the variables in it are forgotten.
7. A return value is the value that a function call evaluates to. Like any value, a return value can be used as part of an expression.
8. If there is no return statement for a function, its return value is `None`.
9. A `global` statement will force a variable in a function to refer to the global variable.
10. The data type of `None` is `NoneType`.
11. That `import` statement imports a module named `areallyourpetsnamederic`. (This isn't a real Python module, by the way.)
12. This function can be called with `spam.bacon()`.
13. Place the line of code that might cause an error in a `try` clause.
14. The code that could potentially cause an error goes in the `try` clause. The code that executes if an error happens goes in the `except` clause.

Chapter 4

1. The empty list value, which is a list value that contains no items. This is similar to how `''` is the empty string value.
2. `spam[2] = 'hello'` (Notice that the third value in a list is at index `2` because the first index is `0`.)
3. `'d'` (Note that `'3' * 2` is the string `'33'`, which is passed to `int()` before being divided by `11`. This eventually evaluates to `3`. Expressions can be used wherever values are used.)
4. `'d'` (Negative indexes count from the end.)
5. `['a', 'b']`
6. `1`
7. `[3.14, 'cat', 11, 'cat', True, 99]`
8. `[3.14, 11, 'cat', True]`

9. The operator for list concatenation is `+`, while the operator for replication is `*`. (This is the same as for strings.)
10. While `append()` will add values only to the end of a list, `insert()` can add them anywhere in the list.
11. The `del` statement and the `remove()` list method are two ways to remove values from a list.
12. Both lists and strings can be passed to `len()`, have indexes and slices, be used in `for` loops, be concatenated or replicated, and be used with the `in` and `not in` operators.
13. Lists are mutable; they can have values added, removed, or changed. Tuples are immutable; they cannot be changed at all. Also, tuples are written using parentheses, `(` and `)`, while lists use the square brackets, `[` and `]`.
14. `(42,)` (The trailing comma is mandatory.)
15. The `tuple()` and `list()` functions, respectively
16. They contain references to list values.
17. The `copy.copy()` function will do a shallow copy of a list, while the `copy.deepcopy()` function will do a deep copy of a list. That is, only `copy.deepcopy()` will duplicate any lists inside the list.

Chapter 5

1. Two curly brackets: `{}`
2. `{'foo': 42}`
3. The items stored in a dictionary are unordered, while the items in a list are ordered.
4. You get a `KeyError` error.
5. There is no difference. The `in` operator checks whether a value exists as a key in the dictionary.
6. `'cat' in spam` checks whether there is a `'cat'` key in the dictionary, while `'cat' in spam.values()` checks whether there is a value `'cat'` for one of the keys in `spam`.
7. `spam.setdefault('color', 'black')`
8. `pprint.pprint()`

Chapter 6

1. Escape characters represent characters in string values that would otherwise be difficult or impossible to type into code.
2. `\n` is a newline; `\t` is a tab.
3. The `\\` escape character will represent a backslash character.

4. The single quote in `Howl's` is fine because you've used double quotes to mark the beginning and end of the string.
5. Multiline strings allow you to use newlines in strings without the `\n` escape character.
6. The expressions evaluate to the following:
   - `'e'`
   - `'Hello'`
   - `'Hello'`
   - `'lo world!`
7. The expressions evaluate to the following:
   - `'HELLO'`
   - `True`
   - `'hello'`
8. The expressions evaluate to the following:
   - `['Remember,', 'remember,', 'the', 'fifth', 'of', 'November.']`
   - `'There-can-be-only-one.'`
9. The `rjust()`, `ljust()`, and `center()` string methods, respectively
10. The `lstrip()` and `rstrip()` methods remove whitespace from the left and right ends of a string, respectively.

I want to Thank Dr. AI Sweigart, who write his Original book which called " AUTOMATE THE BORING STUFF WITH PYTHON :

Practical Programming for total Beginners.", which considers the main and only reference of this book.

Dear Students, you can find a simple explanation for the first lessons of this book at the following Link:

https://www.youtube.com/watch?v=1F_OgqRuSdI&list=PL0-84-yl1fUnRuXGFe_F7qSH1LEnn9LkW