SECOND EDITION

PYTHON FOR KIDS

A PLAYFUL INTRODUCTION TO PROGRAMMING

JASON R. BRIGGS

no starch press

PYTHON FOR KIDS

2ND EDITION

A PLAYFUL INTRODUCTION TO PROGRAMMING

by Jason R. Briggs

no starch press

San Francisco

PYTHON FOR KIDS, 2ND EDITION. Copyright © 2023 by Jason R. Briggs.

All rights reserved. No part of this work may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the publisher.

26 25 24 23 22   1 2 3 4 5
ISBN-13: 978-1-7185-0302-1 (print)
ISBN-13: 978-1-7185-0303-8 (ebook)

Publisher: William Pollock
Managing Editor: Jill Franklin
Production Editor: Paula Williamson
Developmental Editors: William Pollock, Eva Morrow, Jill Franklin
Cover and Interior Design: Octopod Studios
Illustrator: Miran Lipovača
Technical Reviewer: Daniel Zingaro
Copyeditor: George Hale
Proofreader: Sharon Wilkey

For information on distribution, bulk sales, corporate sales, or translations, please contact No Starch Press, Inc. directly at info@nostarch.com or:

No Starch Press, Inc.
245 8th Street, San Francisco, CA 94103
phone: 415.863.9900
www.nostarch.com

The Library of Congress has catalogued the first edition as follows
```
Briggs, Jason R.
Python for kids : a playful introduction to programming / by Jason R. Briggs.
pages cm
Audience: 10+
Includes index.
ISBN 978-1-59327-407-8 -- ISBN 1-59327-407-6
1. Computer software--Juvenile literature. 2. Computer programming--Juvenile
literature. 3. Python
(Computer program language)--Juvenile literature. I. Title.
QA76.52.B75 2013
005.13’3--dc23
```

No Starch Press and the No Starch Press logo are registered trademarks of No Starch Press, Inc. Other product and company names mentioned herein may be the trademarks of their respective owners. Rather than use a trademark symbol with every occurrence of a trademarked name, we are using the names only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

The information in this book is distributed on an "As Is" basis, without warranty. While every precaution has been taken in the preparation of this work, neither the author nor No Starch Press, Inc. shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained in it.

The computer programmer is a creator of universes for which she or he alone is the lawgiver.
No playwright, no stage director, no emperor, however powerful, has ever exercised such absolute authority to arrange a stage or field of battle and to command such unswervingly dutiful actors or troops.

—Joseph Weizenbaum (slightly modified)

ABOUT THE AUTHOR

Jason R. Briggs has been programming since the age of eight, when he first learned BASIC on a Radio Shack TRS-80. He has written software professionally as a developer and technical architect and served as contributing editor for *Java Developer's Journal* . His articles have appeared in *JavaWorld* , *ONJava* , and *ONLamp* . *Python for Kids* (No Starch Press, 2012) is his first book. Jason can be reached at *http://jasonrbriggs.com/* or by email at *mailto:jason@briggs.nz* .

ABOUT THE ILLUSTRATOR

Miran Lipovača is the author of *Learn You a Haskell for Great Good!* . He enjoys boxing, playing bass guitar, and, of course, drawing. He has a fascination with dancing skeletons and the number 71, and when he walks through automatic doors, he pretends that he's actually opening them with his mind.

ABOUT THE TECHNICAL REVIEWER

Dr. Daniel Zingaro is an associate teaching professor of computer science and award-winning teacher at the University of Toronto. His research focuses on understanding and enhancing student learning of computer science. He is the author of two recent No Starch books: *Algorithmic Thinking* (a no-nonsense, no-math guide to algorithms and data structures) and *Learn to Code by Solving Problems* (a primer for learning Python and computational thinking).

FIRST EDITION TECHNICAL REVIEWERS

A recent graduate of The Nueva School, 15-year-old Josh Pollock is a freshman at Lick-Wilmerding High School in San Francisco. He first started programming in Scratch when he was 9 years old, began using TI-BASIC when he was in 6th grade, and moved on to Java and Python in 7th and UnityScript in 8th. In addition to programming, he loves playing the trumpet, developing computer games, and teaching people about interesting STEM topics.

Maria Fernandez has a master's degree in applied linguistics and has been interested in computers and technology for more than 20 years. She taught English to young refugee women with the Global Village Project in Georgia and currently resides in Northern California working with the Educational Testing Service.

BRIEF CONTENTS

CONTENTS IN DETAIL

ACKNOWLEDGMENTS

Thanks to the No Starch team for all their efforts, and to Miran Lipovača for his brilliant illustrations.

Thanks to my wife, daughter, and son, for suffering with a husband and father who had his nose buried in a computer screen even more than usual.

To Mum, for endless amounts of encouragement over the years.

Finally, thanks to my Dad for buying a computer back in the 1970s and putting up with someone who wanted to use it as much as he did. None of this would have been possible without him.

INTRODUCTION

Why learn computer programming? Programming fosters creativity, reasoning, and problem solving. The programmer gets the opportunity to create something from nothing, use logic to turn programming constructs into a form that a computer can run, and, when things don't work quite as well as expected, use problem solving to figure out what has gone wrong. Programming is a fun, sometimes challenging (and occasionally frustrating) activity, and the skills learned from it can be useful both in school and at work—even if your career has nothing to do with computers. And, if nothing else, programming is a great way to spend an afternoon when the weather outside is dreary.

WHY PYTHON

Python is an easy-to-learn programming language that has some really useful features for a beginning programmer. The code is quite easy to read when compared to other programming languages, and it has an interactive shell into which you can enter your programs and see them run.

In addition to its simple language structure and an interactive shell with which to experiment, Python has some features that greatly augment the learning process and allow you to put together simple animations for creating your own games. One is the `turtle` module, inspired by Turtle graphics (used by the Logo programming language back in the 1960s) and designed for educational use. Another is the `tkinter` module, an interface for the Tk graphical user interface (GUI) toolkit, which provides a simple way to create programs with slightly more advanced graphics and animation.

HOW TO LEARN TO CODE

Like anything you try for the first time, it's always best to start with the basics, so begin with the first chapters and resist the urge to skip ahead to the later chapters. No one can play an orchestral symphony the first time they pick up an instrument. Student pilots don't start flying a plane before they understand the basic controls. Gymnasts aren't (usually) able to do backflips on their first try. If you jump ahead too quickly, not only will the basic ideas not stick in your head, but you'll also find the content of the later chapters more complicated than it actually is.

As you go through this book, try all of the examples so you can see how they work. Most chapters also include programming puzzles for you to try, which will help improve your programming skills. Remember that the better you understand the basics, the easier it will be to understand more complicated ideas later on. When you find something frustrating or too challenging, here are some things that I find helpful:

1. Break a problem down into smaller pieces. Try to understand what a small piece of code is doing or think about only a small part of a difficult idea (focus on a small piece of code rather than trying to understand the whole thing at once).
2. If that still doesn't help, it might be best to leave it alone for a while. Sleep on it and come back to it another day. This is a good

way to solve many problems, and it can be particularly helpful for computer programmers.

WHO SHOULD READ THIS BOOK

This book is for anyone interested in computer programming, whether that's a child or an adult coming to programming for the first time. If you want to learn how to write your own software rather than just use the programs developed by others, *Python for Kids* is a great place to start.

In the following chapters, you'll find information to help you install Python, start the Python Shell and perform basic calculations, print text on the screen and create lists, and perform simple control flow operations using `if` statements and `for` loops (and learn what `if` statements and `for` loops are!). You'll learn how to reuse code with functions, the basics of classes and objects, and descriptions of some of the many built-in Python functions and modules.

You'll find chapters on both simple and advanced turtle graphics, as well as on using the `tkinter` module to draw on the computer screen. Programming puzzles of varying complexity are at the ends of many chapters, which will help you cement your newfound knowledge by giving you a chance to write small programs by yourself. Once you've built up your fundamental programming knowledge, you'll learn how to write your own games. You'll develop two graphical games and learn about basic collision detection, events, and different animation techniques.

Most of the examples in this book use Python's IDLE (Integrated DeveLopment Environment) Shell. IDLE provides syntax highlighting, copy-and-paste functionality (similar to what you would use in other applications), and an editor window where you can save your code for later use. This means IDLE works as both an interactive environment for experimentation and something a bit like a text editor. The examples will work just as well with the standard console and a regular text editor, but IDLE's syntax highlighting and slightly more user-friendly environment can aid understanding, so the very first chapter shows you how to set it up.

WHAT'S IN THIS BOOK

Here's a brief rundown of what you'll find in each chapter.

Chapter 1 is an introduction to programming with instructions for installing Python for the first time.

Chapter 2 introduces basic calculations and variables, and **Chapter 3** describes some of the basic Python types, such as strings, lists, and tuples.

Chapter 4 is the first taste of the `turtle` module. We'll jump from basic programming to moving a turtle (in the shape of an arrow) around the screen.

Chapter 5 covers the variations of conditions and `if` statements, and **Chapter 6** moves on to `for` loops and `while` loops.

Chapter 7 is where we start to use and create functions, and then in **Chapter 8** we cover classes and objects. We cover enough of the basic ideas to support some of the programming techniques we'll need in the games development chapters later on in the book. At this point, the material starts to get a little more complicated.

Chapter 9 returns to the `turtle` module as you experiments with more complicated shapes. **Chapter 10** moves on to using the `tkinter` module for more advanced graphics creation.

In **Chapters 11** and **12** , we create our first game, *Bounce!* , which builds on the knowledge gained from the preceding chapters, and in **Chapters 13 – 16** , we create another game, *Mr. Stick Man Races for the Exit* . The game development chapters are where things could start to go seriously wrong. If all else fails, download the code from the companion website (*http://python-for-kids.com*), and compare your code with these working examples.

Finally, in the **Afterword** , we briefly look at how to use the Python package installer (`pip`) to install the `PyGame` module and a short `PyGame` example, before seeing some examples of other programming languages.

In **Appendix A** , you'll find a list of the Python keywords, and in **Appendix B** , a list of some of the useful built-in functions (you'll learn what *keywords* and *functions* are later in the book). **Appendix C** provides some troubleshooting information for common problems.

PYTHON FOR KIDS WEBSITE

If you find that you need help as you read, try the website *http://python-for-kids.com* . There, you'll find downloads for all the examples in this book and links to further information, including where you can download the source code used in the book.

HAVE FUN!

Remember as you work your way through this book that programming can be fun. Don't think of this as work. Think of programming as a way to create some fun games or applications that you can share with your friends or others.

Learning to program is a wonderful mental exercise, and the results can be very rewarding. But most of all, whatever you do, have fun!

PART I

LEARNING TO PROGRAM

1
NOT ALL SNAKES SLITHER

A computer program is a set of instructions that causes a computer to perform some kind of action. It isn't the physical parts of a computer —like the wires, microchips, cards, hard drive, and such—but the hidden stuff running on that hardware. A computer program, which I'll usually refer to as just a *program* , is the set of commands that tells that hardware what to do. *Software* is a collection of computer programs.

Without computer programs, almost every device you use daily would either stop working or be much less useful than it is now. Computer programs, in one form or another, control not only your personal computer but also video game systems, mobile phones, and the GPS units in cars. Software also controls items like LCD TVs and their remote controllers, as well as some of the newest radios, DVD players, ovens, and some fridges. Even car engines, traffic lights, street lamps, train signals, electronic billboards, and elevators are controlled by programs.

Programs are a bit like thoughts. If you didn't have thoughts, you would probably just sit on the floor, staring vacantly at a wall. Your thought "get up off the floor" is an instruction, or *command* , that tells your body to stand up. In the same way, computer programs use commands to tell computers what to do.

If you know how to write computer programs, you can do all sorts of useful things. Sure, you may not be able to write programs to control cars, traffic lights, or your fridge (well, at least not at first), but you could create web pages, write your own games, or even make a program to help with homework.

A FEW WORDS ABOUT LANGUAGE

Like humans, computers use multiple languages to communicate—these are called programming languages. A *programming language* is simply a way to talk to a computer by using instructions that both humans and computers can understand.

There are programming languages named after people (like Ada and Pascal), those named using simple acronyms (like BASIC and FORTRAN), and even a few named after TV shows, like Python. Yes, the Python programming language was named after the *Monty Python's Flying Circus* TV show, not after the snake.

NOTE

Monty Python's Flying Circus *was an alternative British comedy show first broadcast in the 1970s, and it remains hugely popular today among a certain audience. The show had sketches like "The Ministry of Silly Walks," "The Fish-Slapping Dance," and "The Cheese Shop" (which didn't sell any cheese).*

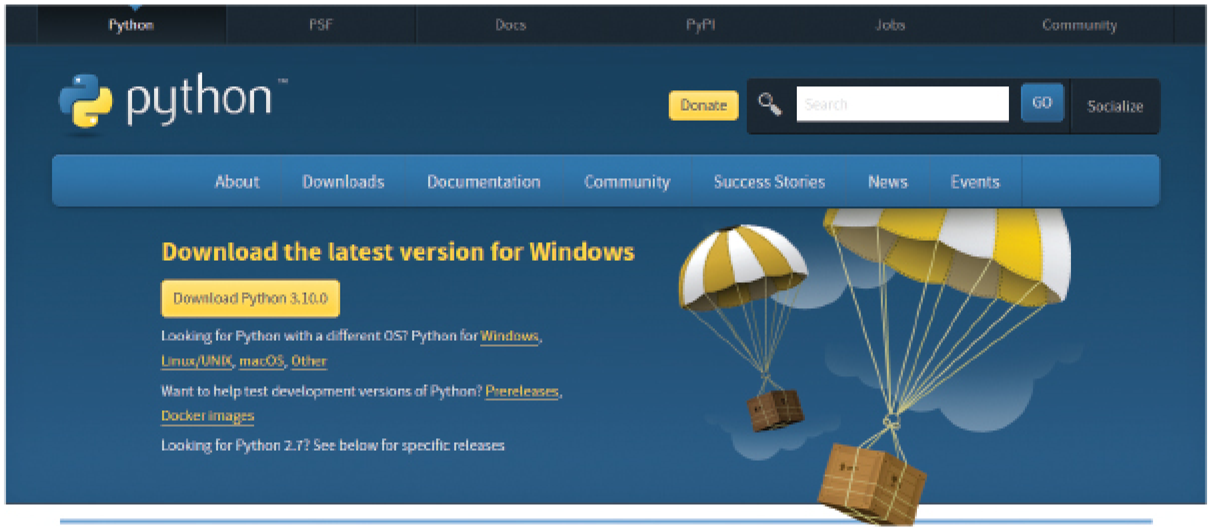
The Python programming language has many features that make it extremely useful for beginners. Most importantly, you can use Python to write simple, efficient programs quite quickly. Python doesn't use as many complicated symbols as other programming languages, which makes it easier to read and a lot friendlier for beginners. (That isn't to say Python doesn't use symbols—they're just not used quite as heavily as in many other languages.)

INSTALLING PYTHON

Installing Python is fairly straightforward. Here, we'll go over the steps for installing it on Windows, macOS, Ubuntu, and Raspberry Pi. When installing Python, you'll also install the IDLE program, which is the **I** ntegrated **D** eve **L** opment **E** nvironment that lets you write programs for Python. If Python has already been installed on your computer, jump ahead to "Once You've Installed Python" on page 10 .

INSTALLING PYTHON ON WINDOWS

To install Python for Microsoft Windows 11, download a version of Python for Windows that is 3.10 or higher at *http://www.python.org/downloads/* . The exact version of Python that you download is not important, as long as it's at least version 3.10. However, if you are using an older version of Windows (like Windows 7), the latest version of Python will not work—in this case, you'll need to install Python 3.8. See the Windows download page (*https://www.python.org/downloads/windows/*) regarding which versions of Python will work with your version of Windows.



Download the latest version for Windows

Looking for Python with a different OS? Python for Windows, Linux/UNIX, macOS, Other

Want to help test development versions of Python? Prereleases, Docker images

Looking for Python 2.7? See below for specific releases

Figure 1-1: The Python download for Windows

If your browser asks whether to save or open your file, choose to save it. Once you've downloaded the Python for Windows installation file, you should be prompted to run it. If not, open your *Downloads* folder and double-click the file. Now, follow the installation instructions on screen to install Python in the default location, as follows:

1. Click **Install Now** .
2. When asked whether to allow the app to make changes to your device, choose **Yes** .
3. Click **Close** once installation finishes, and you should see a number of Python 3.1 *x* icons in your Windows Start menu:

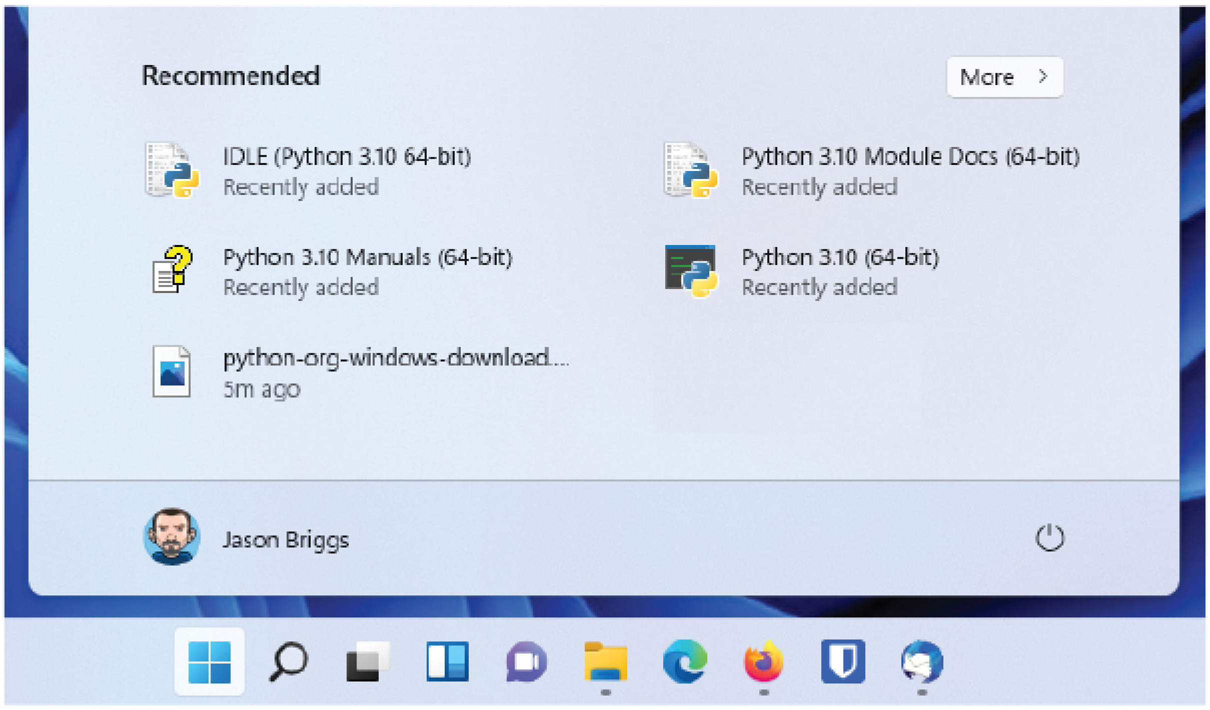

Recommended
More
IDLE (Python 3.10 64-bit)
Recently added
Python 3.10 Module Docs (64-bit)
Recently added
Python 3.10 Manuals (64-bit)
Recently added
Python 3.10 (64-bit)
Recently added
python-org-windows-download...
5m ago
Jason Briggs

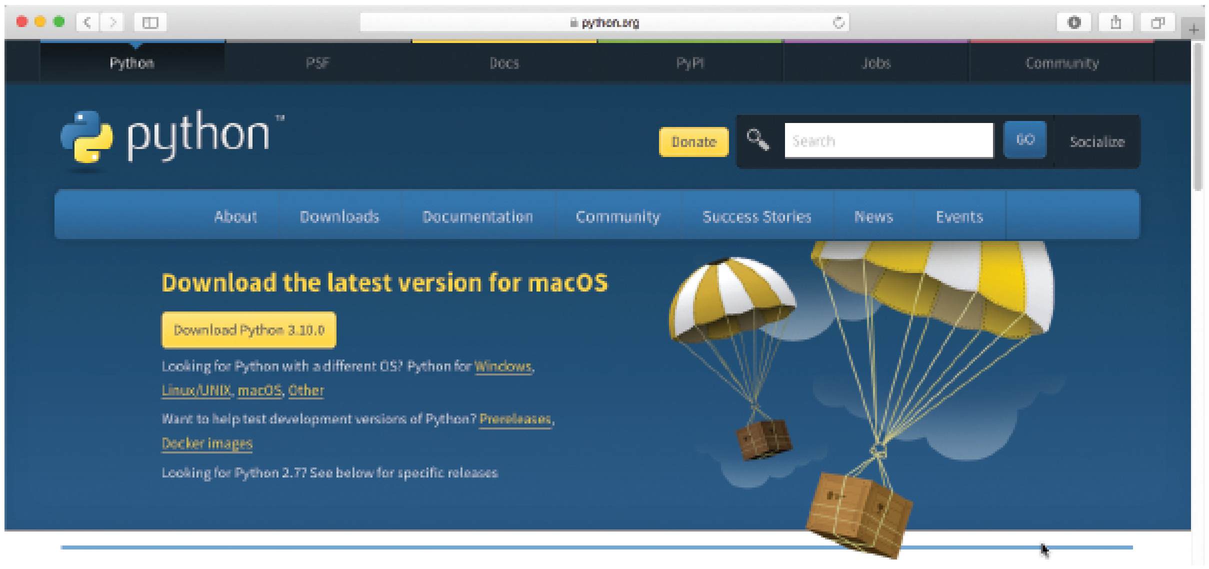
Figure 1-2: Your Start menu may look different depending on the version of Python you use.

Now skip to "Once You've Installed Python" on page 10 to get started with Python.

INSTALLING PYTHON ON MACOS

If you're using a Mac, you should find a version of Python pre-installed, but it might be an older version of the language. To be sure you're running a recent enough version, click the spotlight icon (the magnifying glass in the top-right corner), and type **terminal** in the dialog that appears. When the terminal opens, type `python3 --version` (that's two hyphens, followed by the word *version*) and hit ENTER.

If you see either `command not found` or a version that is less than 3.10, point your browser to the following URL to download the latest installer for macOS: *http://www.python.org/downloads/* .



Looking for Python with a different OS? Python for Windows, Linux/UNIX, macOS, Other

Want to help test development versions of Python? Prereleases, Docker images

Looking for Python 2.7? See below for specific releases

Figure 1-3: Python download for macOS

Once downloaded, double-click the file (it should be called something like *python-3.10.0-macosx11.pkg*). Agree to the license and follow the onscreen instructions to install the software. You should be prompted for the administrator password for your Mac before Python installs. If you don't have the password, ask your parents or whoever is the owner of your machine.

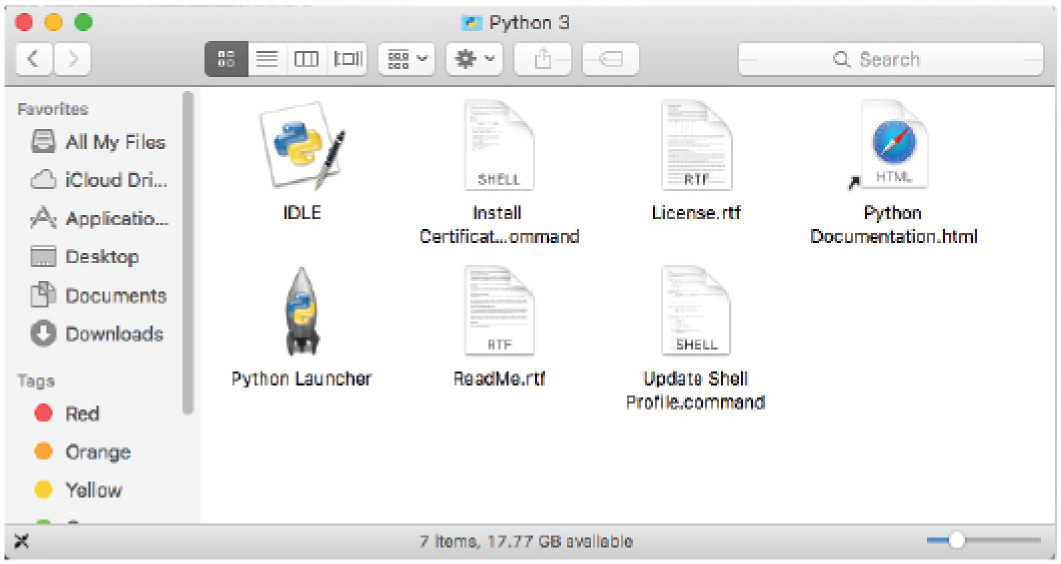


Figure 1-4: Python in Mac Finder

Skip to "Once You've Installed Python" on page 10 to get started with Python.

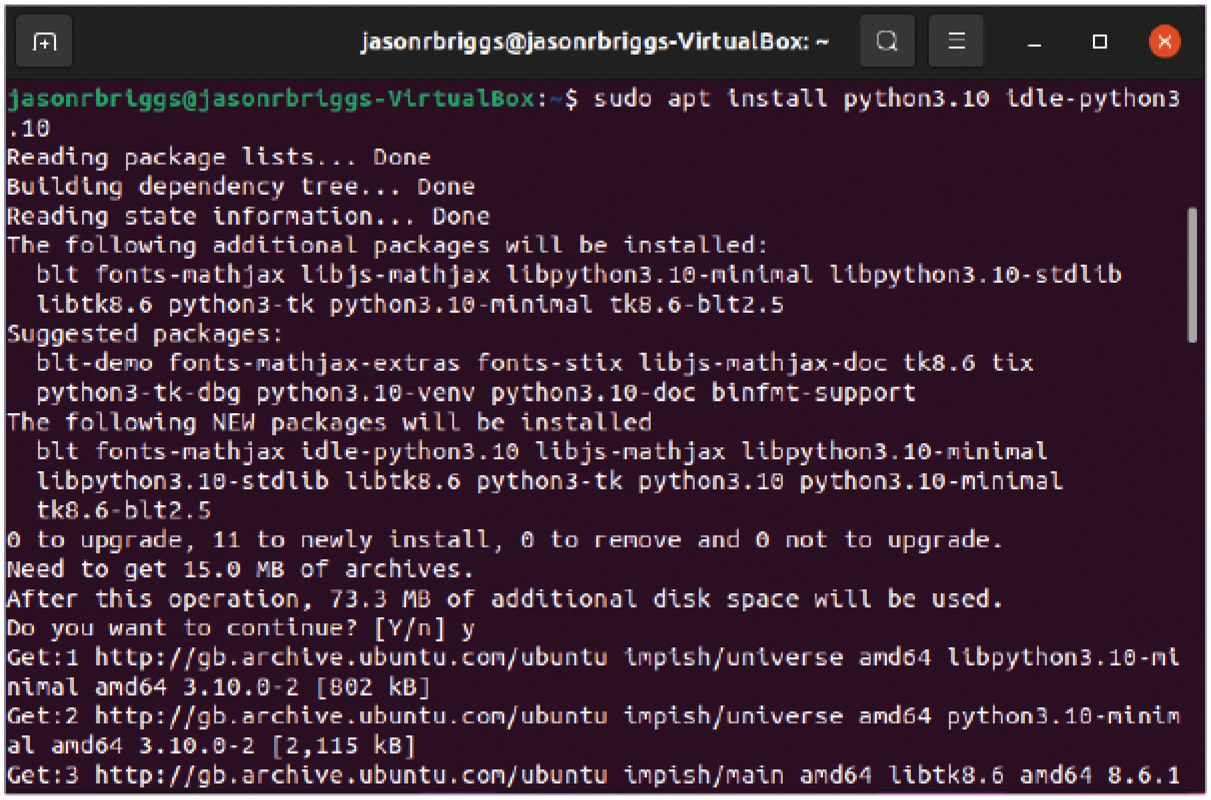
INSTALLING PYTHON ON UBUNTU

Python comes pre-installed on Ubuntu Linux, but it may not be the latest version. Follow these instructions to get the latest version of Python (note that you may need to change the version number in the command that follows to reflect the latest version).

1. Click the Show Applications icon (usually nine dots in the bottom-left corner of the screen).
2. Enter **terminal** in the input box (or click **Terminal** if it's already displayed).
3. When the terminal window appears, enter:

```
sudo apt update
sudo apt install python3.10 idle-python3.10
```

   You might be prompted to enter the administrator password for your computer after entering the first command (if you don't have the administrator password, you may need to ask a parent or teacher to enter it).



```
jasonrbriggs@jasonrbriggs-VirtualBox:~$ sudo apt install python3.10 idle-python3
.10
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
The following additional packages will be installed:
  blt fonts-mathjax libjs-mathjax libpython3.10-minimal libpython3.10-stdlib
  libtk8.6 python3-tk python3.10-minimal tk8.6-blt2.5
Suggested packages:
  blt-demo fonts-mathjax-extras fonts-stix libjs-mathjax-doc tk8.6 tix
  python3-tk-dbg python3.10-venv python3.10-doc binfmt-support
The following NEW packages will be installed
  blt fonts-mathjax idle-python3.10 libjs-mathjax libpython3.10-minimal
  libpython3.10-stdlib libtk8.6 python3-tk python3.10 python3.10-minimal
  tk8.6-blt2.5
0 to upgrade, 11 to newly install, 0 to remove and 0 not to upgrade.
Need to get 15.0 MB of archives.
After this operation, 73.3 MB of additional disk space will be used.
Do you want to continue? [Y/n] y
Get:1 http://gb.archive.ubuntu.com/ubuntu impish/universe amd64 libpython3.10-mi
nimal amd64 3.10.0-2 [802 kB]
Get:2 http://gb.archive.ubuntu.com/ubuntu impish/universe amd64 python3.10-minim
al amd64 3.10.0-2 [2,115 kB]
Get:3 http://gb.archive.ubuntu.com/ubuntu impish/main amd64 libtk8.6 amd64 8.6.1
```

Figure 1-5: Python installation in the Terminal on Ubuntu; your output may look slightly different depending on which version you download

Skip to "Once You've Installed Python" on page 10 to get started with Python.

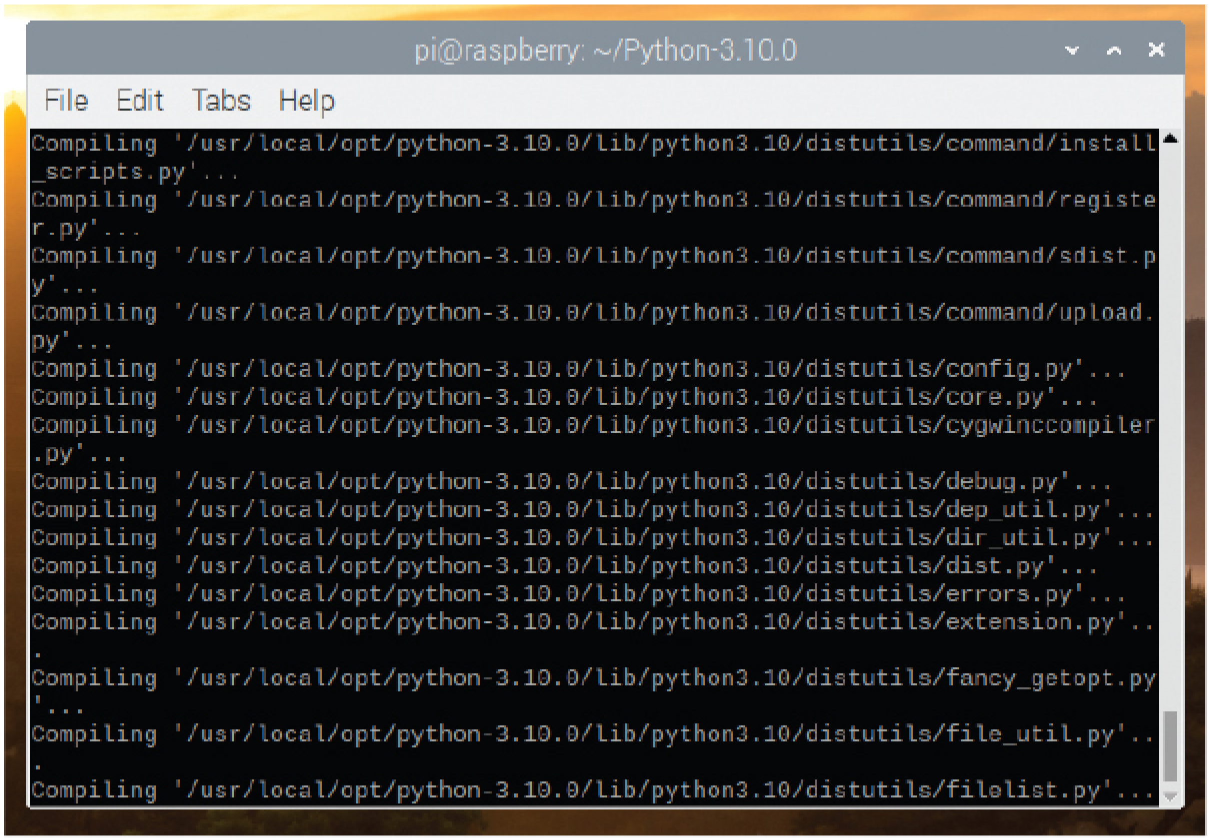
INSTALLING PYTHON ON RASPBERRY PI (RASPBERRY PI OS OR RASPBIAN)

Python 3 comes pre-installed with Raspberry Pi's operating system, but at the time of this writing, it's version 3.7. Installing a later version is slightly more complicated than with the other operating systems—you need to download and build the Python installation yourself. This isn't as scary as it sounds. Simply enter the following commands one by one and wait for each to complete (note that you may need to change the version numbers if you are downloading a version of Python later than 3.10):

```
sudo apt update
sudo apt install libffi-dev libssl-dev tk tk-dev
```

```
wget https://www.python.org/ftp/python/3.10.0/Python-3.10.0.tar.xz
tar -xvf Python3.10.0.tar.xz
cd Python-3.10.0
./configure --prefix=/usr/local/opt/python-3.10.0
make -j 4
sudo make altinstall
```

The second-to-last step will take the longest to finish because it's building all the code that goes into the Python application.



```
Compiling '/usr/local/opt/python-3.10.0/lib/python3.10/distutils/command/install
_scripts.py'...
Compiling '/usr/local/opt/python-3.10.0/lib/python3.10/distutils/command/registe
r.py'...
Compiling '/usr/local/opt/python-3.10.0/lib/python3.10/distutils/command/sdist.p
y'...
Compiling '/usr/local/opt/python-3.10.0/lib/python3.10/distutils/command/upload.
py'...
Compiling '/usr/local/opt/python-3.10.0/lib/python3.10/distutils/config.py'...
Compiling '/usr/local/opt/python-3.10.0/lib/python3.10/distutils/core.py'...
Compiling '/usr/local/opt/python-3.10.0/lib/python3.10/distutils/cygwinccompiler
.py'...
Compiling '/usr/local/opt/python-3.10.0/lib/python3.10/distutils/debug.py'...
Compiling '/usr/local/opt/python-3.10.0/lib/python3.10/distutils/dep_util.py'...
Compiling '/usr/local/opt/python-3.10.0/lib/python3.10/distutils/dir_util.py'...
Compiling '/usr/local/opt/python-3.10.0/lib/python3.10/distutils/dist.py'...
Compiling '/usr/local/opt/python-3.10.0/lib/python3.10/distutils/errors.py'...
Compiling '/usr/local/opt/python-3.10.0/lib/python3.10/distutils/extension.py'..
.
Compiling '/usr/local/opt/python-3.10.0/lib/python3.10/distutils/fancy_getopt.py
'...
Compiling '/usr/local/opt/python-3.10.0/lib/python3.10/distutils/file_util.py'..
.
Compiling '/usr/local/opt/python-3.10.0/lib/python3.10/distutils/filelist.py'...
```

Figure 1-6: Python installation in the Terminal on Raspberry Pi; your output may look slightly different depending on which version of Python you download.

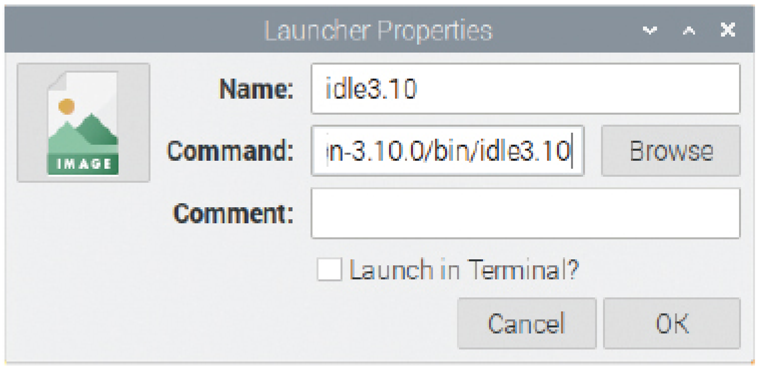
Once Python is installed, you need to add a program called IDLE into the menu (this makes life easier later):

1. Click the raspberry icon in the top left of the screen, then click **Preferences** and **Main Menu Editor** .
2. In the window that appears, click **Programming** , and then click the **New Item** button.

3. In the Launcher Properties dialog shown in Figure 1-7 , enter the name as **idle3.10** , and enter this as the command, changing the version number as needed:

```
/usr/local/opt/python-3.10.0/bin/idle3.10
```

4. Click **OK** , then **OK** again in the main editor window to finish. Then you can move on to the next section.



Name: idle3.10
Command: n-3.10.0/bin/idle3.10
Comment:

Figure 1-7: Launcher set up in the Raspberry Pi

ONCE YOU'VE INSTALLED PYTHON

With Python installed, let's write our first program in IDLE (also called the *Shell*).

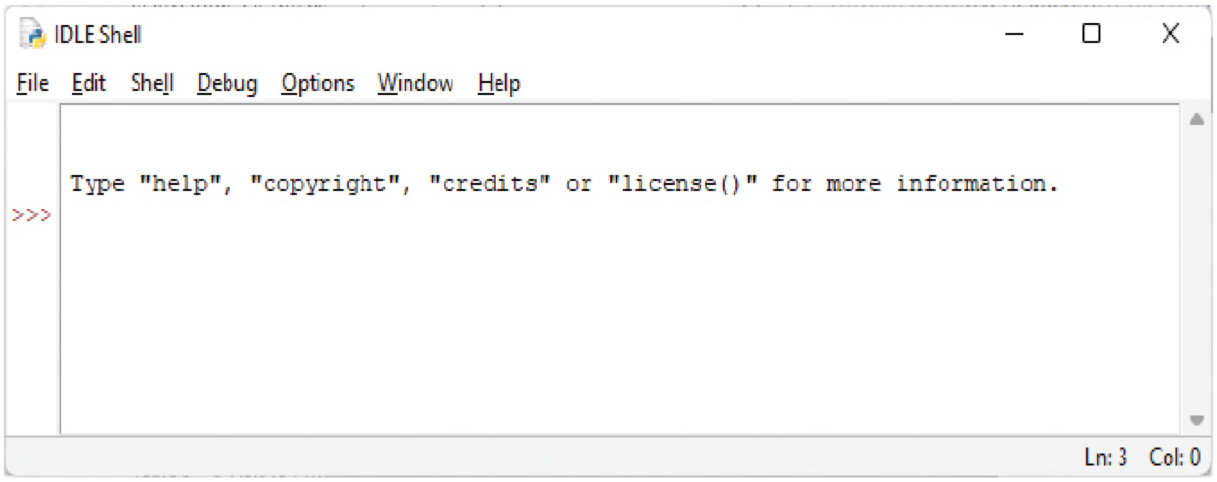
If you're using Windows, enter **idle** in the Windows search box (bottom left of the screen), and select **IDLE (Python 3.1 *x* 64-bit)** when it appears in the Best Match menu.

If you're using a Mac, navigate to **Go** ▸ **Applications** and open the Python 3.1 *x* folder to find IDLE.

If you're using Ubuntu, when you click **Show Applications** and then click the **All** tab at the bottom, you should see an entry titled *IDLE (using Python-3.1* x *)* —you can also enter IDLE in the search box if you can't see it.

If you're using a Raspberry Pi, click the raspberry icon at the top left of the screen, click **Programming** , and then select **idle3.1 *x*** from the list displayed.

When you open IDLE, you should see a window like this:



```
Type "help", "copyright", "credits" or "license()" for more information.
>>>
```

Figure 1-8: IDLE Shell in Windows

This is the *Python Shell* , which is part of Python's integrated development environment. The three greater-than signs (`>>>`) are called the *prompt* .

Let's enter some commands at the prompt, beginning with the following:

```
>>> print("Hello World")
```

Make sure to include the double quotes (" "). Press ENTER on your keyboard when you're finished typing the line. If you've entered the command correctly, you should see something like this:

```
>>> print("Hello World")
Hello World
>>>
```

The prompt should reappear to let you know that the Python Shell is ready to accept more commands.

Congratulations! You've just created your first Python program. The word `print` is a type of Python command called a *function* , and it prints out whatever is inside the parentheses to the screen. In essence, you have given the computer an instruction to display the words “Hello World”—an instruction both you and the computer can understand.

SAVING YOUR PYTHON PROGRAMS

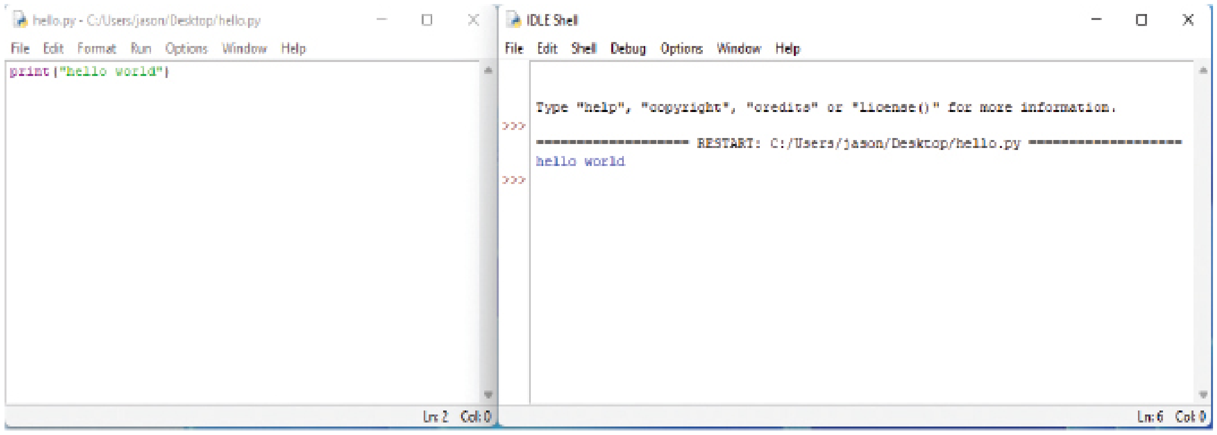
Python programs wouldn’t be very useful if you needed to rewrite them every time you wanted to use them, never mind print them out so you could reference them. Sure, it might be fine to rewrite short programs, but a large program, like a word processor, could contain millions of lines of code. Print that all out, and you could have well over 100,000 pages. Just imagine trying to carry that huge stack of paper home. Better hope that you won’t meet up with a big gust of wind.

Luckily, we can save our programs for future use. To create and save a new program, open IDLE and choose **File** ▸ **New Window** . An empty window will appear, with *Untitled* in the menu bar. Enter the following code into the new shell window:

```
>>> print("Hello World")
```

Now, choose **File** ▸ **Save** . When prompted for a filename, enter *hello.py* , and save the file to your desktop. Then choose **Run** ▸ **Run Module** .

With any luck, your saved program should run, like this:


hello.py - C:/Users/jason/Desktop/hello.py
File Edit Format Run Options Window Help
Ln: 2 Col: 0
IDLE Shell
File Edit Shell Debug Options Window Help
Ln: 6 Col: 0

```
print("hello world")
```

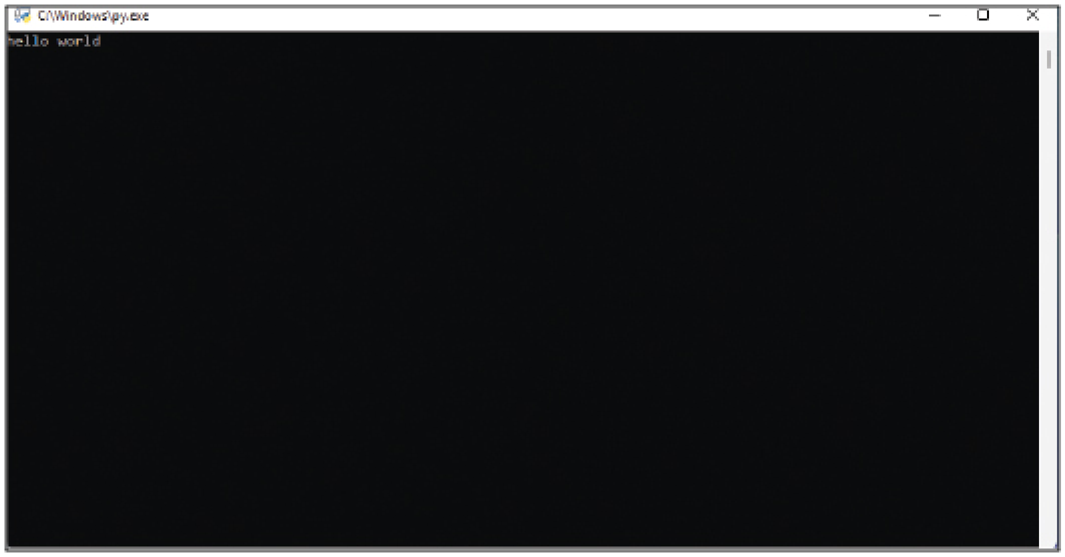
```
Type "help", "copyright", "credits" or "license()" for more information.
>>>
================== RESTART: C:/Users/jason/Desktop/hello.py ==================
hello world
>>>
```

Figure 1-9: Hello World in IDLE

Now, if you close the shell window but leave the *hello.py* window open, and then choose **Run** ▸ **Run Module** , the Python Shell should reappear, and your program should run again (to reopen the Python Shell without running the program, choose **Run** ▸ **Python Shell**).

After running the code, you'll find a new icon on your desktop labeled *hello.py* . If you double-click the icon, a black window will appear briefly and then vanish. What happened?

You're seeing the Python command line console (similar to the shell) start up, print `Hello World` , and then exit. Here's what would appear if you had superhero-like speed vision and could see the window before it closed:



```
hello world
```

Figure 1-10: Hello World in the console

NOTE

Depending on your operating system, this might not work—or it may run using a different version of Python than the one we've installed.

In addition to the menus, you can use keyboard shortcuts to create a new shell window, save a file, and run a program:

- On Windows, Ubuntu, and Raspberry Pi, press CTRL-N to create a new shell window, CTRL-S to save your file after you've finished editing, and F5 to run your program.
- On macOS, press COMMAND-N to create a new shell window, COMMAND-S to save your file, and hold down the function (Fn) key and press F5 to run your program.

WHAT YOU LEARNED

We began simply in this chapter with a Hello World application—the program nearly everyone starts with when they learn computer programming. In the next chapter, we'll do some more useful things with the Python Shell.

2
CALCULATIONS AND VARIABLES

Now that you've installed Python and know how to start the Python Shell, you're ready to do something with it. We'll begin with some simple calculations and then learn how to use variables. Variables are a way of storing things in a computer program, and they can help you write useful programs.

CALCULATING WITH PYTHON

Normally, when asked to find the product of two numbers like 8 × 3.57, you would use a calculator or a pencil and paper. Well, how about using the Python Shell to perform your calculation? Let's try it.

Start the Python Shell by double-clicking the IDLE icon on your desktop, or if you're using Ubuntu, click the IDLE icon in the Applications menu. At the prompt, enter this calculation:

```
>>> 8 * 3.57
28.56
```

When entering a multiplication calculation in Python, you use the asterisk symbol (*) instead of a multiplication sign (×).

How about we try an equation that's a bit more useful?

Suppose you are digging in your backyard and uncover a bag of 20 gold coins. The next day, you sneak down to the basement and stick the coins inside your grandfather's steam-powered replicating invention (luckily, you can *just* fit the 20 coins inside). You hear a whiz and a pop and, a few hours later, out shoot another 10 gleaming coins.

How many coins would you have in your treasure chest if you did this every day for a year? On paper, the equations might look like this:

$$10 \times 365 = 3650$$
$$3650 + 20 = 3670$$

Sure, it's easy enough to do these calculations using a calculator or on paper, but we can do all of these calculations with the Python Shell as well. First, we multiply 10 coins by 365 days in a year to get 3650. Next, we add the original 20 coins to get 3670.

```
>>> 10 * 365
3650
>>> 3650 + 20
3670
```

Now, what if a raven spots the shiny gold sitting in your bedroom, and every week flies in and manages to steal three coins? How many coins would you have left at the end of the year? Here's how this calculation looks in the Python Shell:

```
>>> 3 * 52
156
>>> 3670 - 156
3514
```

First, we multiply 3 coins by 52 weeks in the year. The result is 156. We subtract that number from our total number of coins (3670), which tells us that we would have 3514 coins remaining at the end of the year.

Although you could easily do this calculation with a calculator, working through it in the Python Shell is beneficial in learning to write simple computer programs. In this book, you'll learn how to expand on these ideas to write programs that are even more useful.

PYTHON OPERATORS

You can do addition, subtraction, multiplication, and division in the Python Shell, among other mathematical operations that we'll explore later on. The basic symbols Python uses to perform mathematical operations, called *operators* , are listed in Table 2-1 .

Table 2-1: Basic Python Operators

Symbol	Operation
+	Addition
-	Subtraction
*	Multiplication
/	Division

The *forward slash* (`/`) is used for division because it's similar to the division line that you would use when writing a fraction. For example, if you had 100 pirates and 20 large barrels and you wanted to calculate how many pirates you could hide in each barrel, you could divide 100 pirates by 20 barrels (100 ÷ 20) by entering `100 / 20` in the Python Shell. Just remember that the forward slash is the one whose top falls to the right.

THE ORDER OF OPERATIONS

We use parentheses in programming languages to control the order of operations. An *operation* is anything that uses an operator. Multiplication and division have a higher order than addition and subtraction, so they're performed first. In other words, if you enter an equation in Python, multiplication or division is performed before addition or subtraction.

For example, in the following equation, the numbers 30 and 20 are multiplied first, and the number 5 is added to their product:

```
>>> 5 + 30 * 20
605
```

This equation is another way of saying, "Multiply 30 by 20, and then add 5 to the result." The result is 605. We can change the order of operations by adding parentheses around the first two numbers, like so:

```
>>> (5 + 30) * 20
700
```

The result of this equation is 700 (not 605) because the parentheses tell Python to do the operation in the parentheses first, and then do the operation outside the parentheses. This example is saying, "Add 5 to 30, and then multiply the result by 20."

Parentheses can be *nested* , which means that there can be parentheses inside parentheses, like this:

```
>>> ((5 + 30) * 20) / 10
70.0
```

In this case, Python evaluates the innermost parentheses first, then the outer ones, and then the final division operator. In other words, this equation is saying, "Add 5 to 30, then multiply the result by 20, and divide that result by 10." Here's what happens:

1. Adding 5 to 30 gives 35.
2. Multiplying 35 by 20 gives 700.
3. Dividing 700 by 10 gives the final answer of 70.

If we had not used parentheses, the result would be slightly different:

```
>>> 5 + 30 * 20 / 10
65.0
```

In this case, 30 is first multiplied by 20 (giving 600), and then 600 is divided by 10 (giving 60). Finally, 5 is added to get the result of 65.

NOTE

Remember that multiplication and division always go before addition and subtraction, unless parentheses are used to control the order of operations.

VARIABLES ARE LIKE LABELS

The word *variable* in programming describes a place to store information such as numbers, text, lists of numbers and text, and so on. A variable is essentially a label for something.

For example, to create a variable named `fred` , we use an equal sign (`=`) and then tell Python what information the variable should be the label for. Here, we create the variable `fred` and tell Python that it's a label for the number 100 (note that this doesn't mean that another variable can't have the same value):

```
>>> fred = 100
```

To find the value of a variable, enter `print` in the Python Shell, followed by the variable name in parentheses, like this:

```
>>> print(fred)
100
```

We can also tell Python to change the variable `fred` so that it labels something else. For example, here's how to change `fred` to the number 200:

```
>>> fred = 200
>>> print(fred)
200
```

On the first line, we say that `fred` labels a number `200` . In the second line, we print the value of `fred` , just to confirm the change. Python prints the result on the last line.

We can also use more than one label (or variable) for the same item:

```
>>> fred = 200
>>> john = fred
>>> print(john)
200
```

In this example, we're telling Python that we want the name (or variable) `john` to label the same thing as `fred` by using the equal sign between `john` and `fred` .

Of course, `fred` probably isn't a very useful name for a variable because it most likely doesn't tell us anything about what the variable is used for. Let's call our variable `number_of_coins` instead of `fred` , like this:

```
>>> number_of_coins = 200
>>> print(number_of_coins)
200
```

This makes it clear that we're talking about 200 coins. Variable names can be made up of letters, numbers, and the underscore character (`_`), but they can't start with a number. You can use anything from single letters (such as `a`) to long sentences for variable names. (A variable can't contain a space, so use an underscore to separate words.) Sometimes, if you're doing something quick, a short variable name is best. The name you choose should depend on how meaningful you need the variable name to be.

Now that you know how to create variables, let's look at how to use them.

USING VARIABLES

Remember our equation for figuring out how many coins you would have at the end of the year if you could magically create new coins with your grandfather's mysterious invention in the basement? We had this equation:

```
>>> 20 + 10 * 365
3670
>>> 3 * 52
156
>>> 3670 - 156
3514
```

We can turn that into a single line of code:

```
>>> 20 + 10 * 365 - 3 * 52
3514
```

That's not very easy to read, but what if we turn the numbers into variables? Try entering the following:

```
>>> found_coins = 20
>>> magic_coins = 10
>>> stolen_coins = 3
```

These entries create the variables `found_coins` , `magic_coins` , and `stolen_coins` .

Now, we can reenter the equation like this:

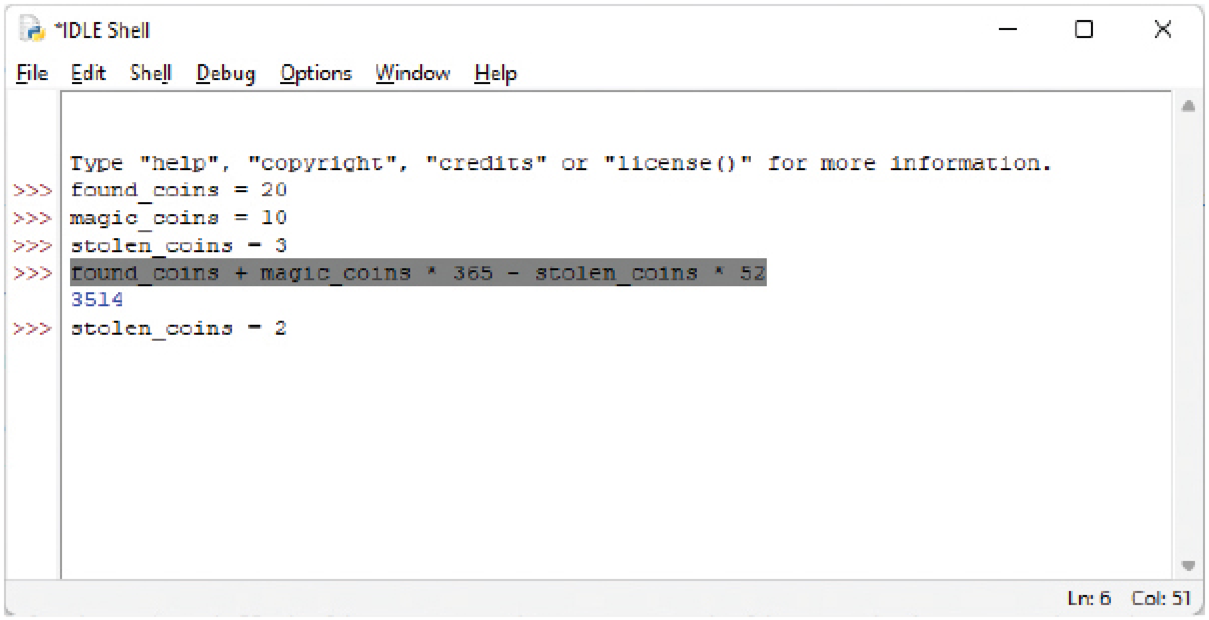
```
>>> found_coins + magic_coins * 365 - stolen_coins * 52
3514
```

You can see that this gives us the same answer. So who cares, right? Ah, but here's the magic of variables. What if you stick a scarecrow in your window, and the raven steals only two coins instead of three? When we use a variable, we can simply change the variable to hold that new number, and it will change everywhere it is used in the equation. We can change the `stolen_coins` variable to `2` by entering this:

```
>>> stolen_coins = 2
```

We can then copy and paste the equation to calculate the answer again, like so:

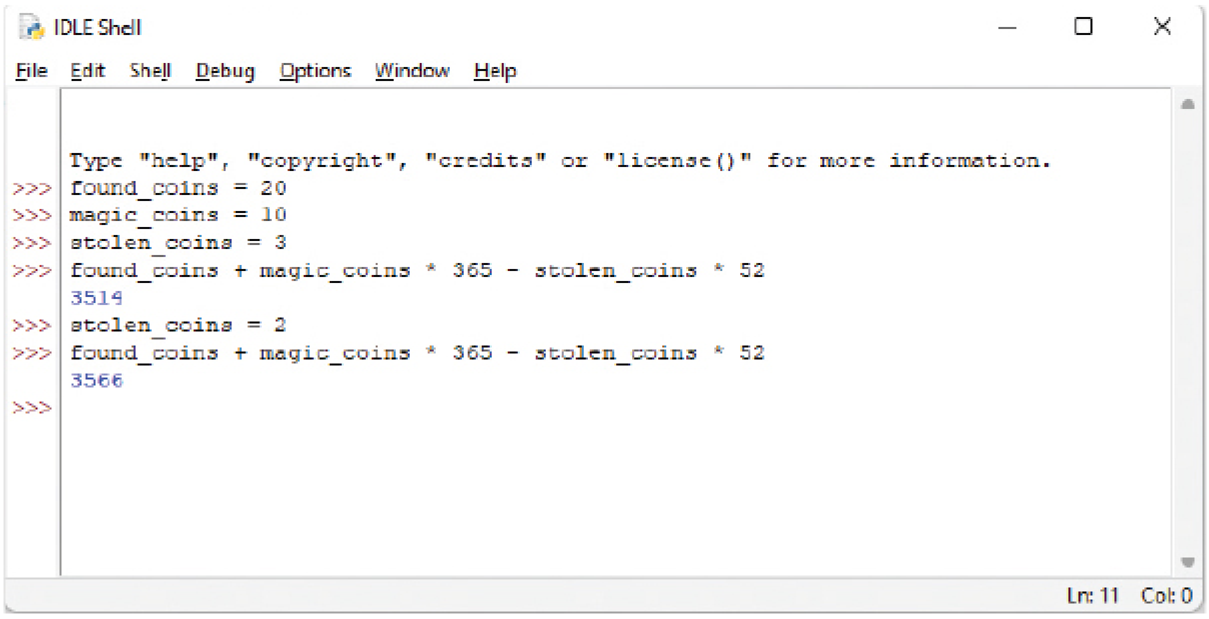
1. Select the text to copy by clicking with the mouse and dragging from the beginning to the end of the line, as shown in Figure 2-1 .



```
Type "help", "copyright", "credits" or "license()" for more information.
>>> found_coins = 20
>>> magic_coins = 10
>>> stolen_coins = 3
>>> found_coins + magic_coins * 365 - stolen_coins * 52
3514
>>> stolen_coins = 2
```

Figure 2-1: Selecting in the Python Shell

1. Hold down the CTRL key (or, if you're using a Mac, the COMMAND key ⌘) and press C to copy the selected text. (You'll see this as CTRL-C from now on.)
2. Click the last prompt line (after `stolen_coins = 2`).
3. Hold down the CTRL (or COMMAND) key and press V to paste the selected text. (You'll see this as CTRL-V from now on.)
4. Press ENTER to see the new result.



```
Type "help", "copyright", "credits" or "license()" for more information.
>>> found_coins = 20
>>> magic_coins = 10
>>> stolen_coins = 3
>>> found_coins + magic_coins * 365 - stolen_coins * 52
3514
>>> stolen_coins = 2
>>> found_coins + magic_coins * 365 - stolen_coins * 52
3566
>>>
```

Figure 2-2: Pasting in the Python Shell

That's much easier than retyping the whole equation!

You can try changing the other variables, and then copy (CTRL-C) and paste (CTRL-V) the calculation to see the effect of your changes. For example, if you bang the sides of your grandfather's invention at the right moment, and it spits out three extra coins each time, you'll find that you end up with 4661 coins at the end of the year:

```
>>> magic_coins = 13
>>> found_coins + magic_coins * 365 stolen_coins * 52
4661
```

Of course, using variables for a simple equation like this one is still only *slightly* useful. We haven't gotten to *really* useful yet. For now, just remember that variables are a way of labeling things so that you can use them later.

WHAT YOU LEARNED

In this chapter, you learned how to do simple equations using Python operators and how to use parentheses to control the order of operations (the order in which Python evaluates the parts of the equations). We then created variables to label values and used those variables in our calculations.

3
STRINGS, LISTS, TUPLES, AND DICTIONARIES

In Chapter 2 , we did some basic calculations with Python and learned about variables. In this chapter, we'll work with other items in Python programs: strings, lists, tuples, and dictionaries. You'll use strings to display messages in your programs (such as "Get Ready" and "Game Over" messages in a game). You'll also discover how lists, tuples, and dictionaries are used to store collections of things.

STRINGS

When programming, we usually call text a *string* . Think of a string as a collection of letters. All the letters, numbers, and symbols in this book could be a string, and so could your name and address. In fact, the first Python program we created in Chapter 1 used a string: "Hello World."

CREATING STRINGS

In Python, we create a string by putting quotes around text because programming languages need to distinguish between different types of values. (We need to tell the computer whether a value is a number, a string, or something else.) For example, we could take our `fred` variable from Chapter 2 and use it to label a string:

```
fred = "Why do gorillas have big nostrils? Big fingers!!"
```

Then, to see what's "inside" `fred` , we could enter `print(fred)` :

```
>>> print(fred)
Why do gorillas have big nostrils? Big fingers!!
```

You can also use single quotes to create a string, like this:

```
>>> fred = 'What is pink and fluffy? Pink fluff!!'
>>> print(fred)
What is pink and fluffy? Pink fluff!!
```

However, if you try to enter more than one line of text for your string using only a single (`'`) or double quote (`"`) or if you start with one type of quote and finish with another, you'll get an error message in the Python Shell. For example, enter the following line:

```
>>> fred = "How do dinosaurs pay their bills?
```

You'll see this result:

```
SyntaxError: EOL while scanning string literal
```

This is an error message complaining about syntax because you did not follow the rules for ending a string with a single or double quote.

Syntax means the arrangement and order of words in a sentence or, in this case, the arrangement and order of words and symbols in a program. So *SyntaxError* means that you did something in an order Python was not expecting, or Python was expecting something that you missed. *EOL* means *end-of-line* , so the rest of the error message is telling you that Python hit the end of the line and did not find a double quote to close (or finish) the string.

To use more than one line of text in your string (called a *multiline string*), use three single quotes ('''), and then hit ENTER between lines, like this:

```
>>> fred = '''How do dinosaurs pay their bills?
    With tyrannosaurus checks!'''
```

Let's print the contents of `fred` to see if this worked:

```
>>> print(fred)
How do dinosaurs pay their bills?
With tyrannosaurus checks!
```

HANDLING PROBLEMS WITH STRINGS

Now consider this silly example of a string, which causes Python to display an error message:

```
>>> silly_string = 'He said, "Aren't can't shouldn't wouldn't."'
SyntaxError: invalid syntax
```

In the first line, we try to create a string (defined as the variable `silly_string`) enclosed by single quotes, but also containing a mixture of single quotes in the words `can’t` , `shouldn’t` , and `wouldn’t` , as well as double quotes. What a mess!

Remember that Python is not as smart as a human being, so all it sees is a string containing `He said, "Aren` , followed by a bunch of other characters that it doesn’t expect. When Python sees a quotation mark (either a single or double quote), it expects a string to start following the first quotation mark and the string to end after the next matching quotation mark (either single or double) on that line. In this case, the start of the string is the single quotation mark before `He` , and the end of the string, as far as Python is concerned, is the single quote after the `n` in `Aren` .

In the last line, Python tells us what sort of error occurred—in this example, a syntax error.

Using double instead of single quotes still produces an error:

```
>>> silly_string = "He said, "Aren't can't shouldn't wouldn't.""
SyntaxError: invalid syntax
```

Here, Python sees a string bound by double quotes, containing the letters `He said,` (and a space). Everything following that string (from `Aren’t` on) causes the error.

This is because, from Python’s perspective, all that extra stuff isn’t supposed to be there. Python looks for the next matching quote and doesn’t know what you want it to do with anything that follows that quote on the same line.

The solution to this problem is a multiline string, which we learned about earlier, using *three* single quotes (`'''`). This allows us to combine double and single quotes in our string without causing errors. In fact, if we use three single quotes, we can put any combination of single and double quotes inside the string (as long as we don’t try to put three single quotes there). The error-free version of our string looks like this:

```
silly_string = '''He said, "Aren't can't shouldn't wouldn't."'''
```

But wait, there's more. If you really want to use single or double quotes to surround a string in Python, instead of three single quotes, you can add a backslash (\) before each quotation mark within the string. This is called *escaping* . It's a way of telling Python, "Yes, I know I have quotes inside my string, and I want you to ignore them until you see the end quote."

Escaping strings can make them harder to read, so it's considered better practice to use multiline strings. Still, you might come across code that uses escaping, so it's helpful to know why the backslashes are there.

Here are a few examples of how escaping works:

```
❶ >>> single_quote_str = 'He said, "Aren\'t can\'t shouldn\'t
     wouldn\'t."'
❷ >>> double_quote_str = "He said, \"Aren't can't shouldn't
     wouldn't.\""
  >>> print(single_quote_str)
  He said, "Aren't can't shouldn't wouldn't."
  >>> print(double_quote_str)
  He said, "Aren't can't shouldn't wouldn't."
```

First, at ❶ , we create a string with single quotes, using the backslash in front of the single quotes inside that string. At ❷ , we create a string with double quotes, and use the backslash in front of those quotes in the string. In the lines that follow, we print the variables we've just created. Notice that the backslash character doesn't appear in the strings when we print them.

EMBEDDING VALUES IN STRINGS

If you want to display a message using the contents of a variable, you can embed it in a special string, called an *f-string* (also known as a *formatted string literal*). You put braces around the variable name, which will then be replaced by the actual value. (*Embedding values* , also referred to as *string substitution* , is programmer-speak for "inserting values.")

For example, to have Python calculate or store the number of points you scored in a game, and then add it to a sentence like "I scored 10 points," add an `f` before the first quote and then replace the number 10 with the variable surrounded by braces `{}` , like this:

```
>>> myscore = 1000
>>> message = f'I scored {myscore} points'
>>> print(message)
I scored 1000 points
```

Here, we create the variable `myscore` with the value `1000` and the variable `message` that contains the f-string `'I scored {myscore} points'` . On the next line, we call `print(message)` to see the result of our string substitution. The result of printing this message is `I scored 1000 points` . We don't need to use a variable for the message. We could do the same example and just use this:

```
print(f'I scored {myscore} points')
```

We can also use more than one variable in a string:

```
>>> first = 0
>>> second = 8
>>> print(f'What did the number {first} say to the number {second}? Nice belt!!')
What did the number 0 say to the number 8? Nice belt!!
```

We can even put expressions in an f-string, like this:

```
>>> print(f'Two plus two equals {2 + 2}')
Two plus two equals 4
```

In this example, Python evaluates the simple equation between the braces, so the printed string contains the result.

MULTIPLYING STRINGS

What is 10 multiplied by 5? The answer is 50, of course. But what's 10 multiplied by *a* ? Here's Python's answer:

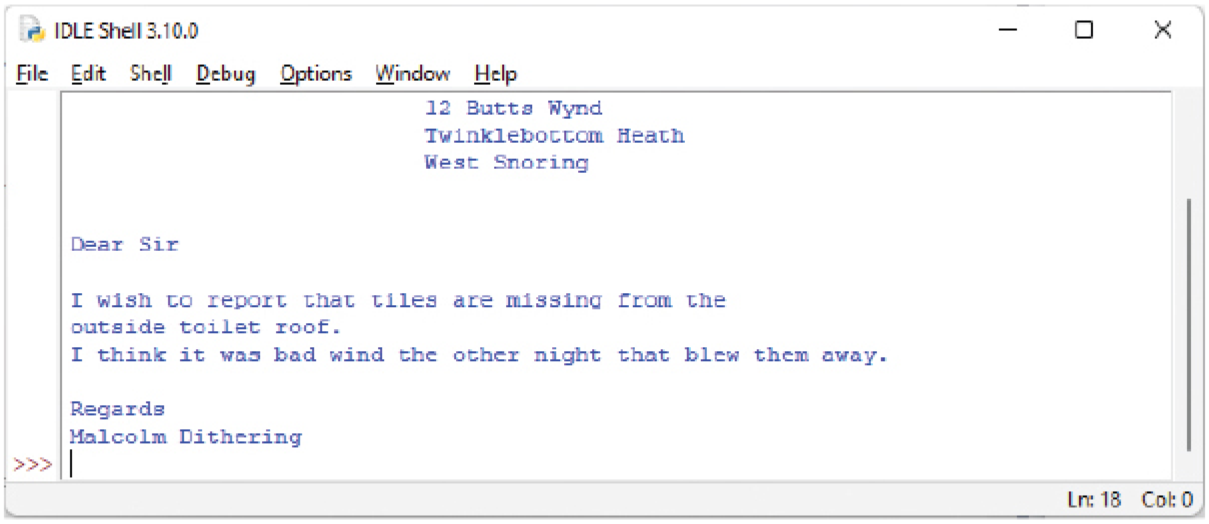
```
>>> print(10 * 'a')
aaaaaaaaaa
```

Python programmers might multiply strings for a number of reasons, such as to line up text with a specific number of spaces when displaying messages in the Python Shell. Try printing the following letter in the Python Shell (select **File** ▸ **New File** , and enter the following code):

```
spaces = ' ' * 25
print('{} 12 Butts Wynd')
print('{} Twinklebottom Heath')
print('{} West Snoring')
print()
print()
print('Dear Sir')
print()
print('I wish to report that tiles are missing from the')
print('outside toilet roof.')
print('I think it was bad wind the other night that blew them away.')
print()
print('Regards')
print('Malcolm Dithering')
```

Once you've typed the code into the Python Shell window, select **File** ▸ **Save As** . Name your file *myletter.py* . You can then run the code (as we've done previously) by selecting **Run** ▸ **Run Module** .

In the first line of this example, we create the variable `spaces` by multiplying a space character by 25. We then use that variable in the next three lines to align the text to the right of the Python Shell. You can see the result of these `print` statements below:



```
                         12 Butts Wynd
                         Twinklebottom Heath
                         West Snoring


Dear Sir

I wish to report that tiles are missing from the
outside toilet roof.
I think it was bad wind the other night that blew them away.

Regards
Malcolm Dithering
>>>
```

Figure 3-1: Running the letter code in the Python Shell

WHAT ARE FILES AND FOLDERS?

A *file* is data (or information) of some kind that can be stored on your computer. Files might include photos, videos, ebooks, and even the school report that you wrote in Microsoft Word.

A *folder* (also called a *directory*) is a collection of other folders and files. When you clicked **Save As** to save your *myletter.py* file, it was stored in a folder.

As we'll see, files and folders are very important in programming.

In addition to using multiplication for alignment, we can also use it to fill the screen with annoying messages. Try the following example:

```
>>> print(1000 * 'snirt')
```

LISTS ARE MORE POWERFUL THAN STRINGS

“Spider legs, toe of frog, bat wing, slug butter, and snake dandruff” is not a very normal shopping list (unless you happen to be a wizard), but we’ll use it as our first example of the differences between strings and lists. We could store this list of items in the `wizard_list` variable by using a string like so:

```
>>> wizard_list = 'spider legs, toe of frog, bat wing, slug butter, snake dandruff'
>>> print(wizard_list)
spider legs, toe of frog, bat wing, slug butter, snake dandruff
```

But we could also create a *list* , a somewhat magical Python object that we can manipulate. Here’s what these items would look like written as a list:

```
>>> wizard_list = ['spider legs', 'toe of frog', 'bat wing',
                   'slug butter', 'snake dandruff']
>>> print(wizard_list)
['spider legs', 'toe of frog', 'bat wing', 'slug butter',
'snake dandruff']
```

Creating a list takes a bit more typing than creating a string, but a list is more useful than a string because items in the list can be easily manipulated. We can print an item in the list by entering a number (called the *index position*) inside square brackets, like this:

```
>>> print(wizard_list[2])
bat wing
```

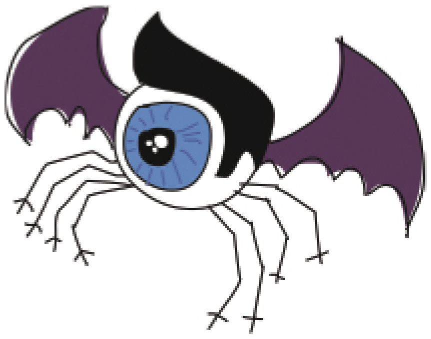
If you were expecting the second item to be `'toe of frog'`, you might be wondering why `'bat wing'` was printed. This is because lists start at index position 0, so the first item in a list is 0, the second is 1, and the **third** is 2. That may not make a lot of sense to humans, but it does to computers.

We can also change an item in a list. Perhaps our wizard friend just let us know that we need to grab snail tongue for them instead of bat wing. Here's how we would change the item in our list:

```
>>> wizard_list[2] = 'snail tongue'
>>> print(wizard_list)
['spider legs', 'toe of frog', 'snail tongue', 'slug butter',
'snake dandruff']
```

This sets the item in index position 2, previously bat wing, to snail tongue.

We can also show a sublist of the items in the list. We do this by using a colon (`:`) inside square brackets. For example, enter the following to see the third to fifth items in our list (a brilliant set of ingredients for a lovely sandwich):



```
>>> print(wizard_list[2:5])
['snail tongue', 'slug butter', 'snake dandruff']
```

Writing `[2:5]` is the same as saying, "Show the items from index position 2 up to (but not including) index position 5"—in other words, items 3, 4, and 5.

Lists can be used to store all sorts of items, like numbers:

```
>>> some_numbers = [1, 2, 5, 10, 20]
```

They can also hold strings:

```
>>> some_strings = ['Which', 'Witch', 'Is', 'Which']
```

They might have mixtures of numbers and strings:

```
>>> numbers_and_strings = ['Why', 'was', 6, 'afraid', 'of', 7,
                           'because', 7, 8, 9]
>>> print(numbers_and_strings)
['Why', 'was', 6, 'afraid', 'of', 7, 'because', 7, 8, 9]
```

And lists might even store other lists:

```
>>> numbers = [1, 2, 3, 4]
>>> strings = ['I', 'kicked', 'my', 'toe', 'and', 'it',
               'is', 'sore']
>>> mylist = [numbers, strings]
>>> print(mylist)
[[1, 2, 3, 4], ['I', 'kicked', 'my', 'toe', 'and', 'it',
'is', 'sore']]
```

This list-within-list example creates three variables: `numbers` with four numbers, `strings` with eight strings, and `mylist` using `numbers` and `strings` . The third list (`mylist`) has only two elements because it's a list of variable names, not the contents of the variables.

We can try printing the two elements of `mylist` separately:

```
>>> print(mylist[0])
[1, 2, 3, 4]
>>> print(mylist[1])
['I', 'kicked', 'my', 'toe', 'and', 'it', 'is', 'sore']
```

Here we can see `mylist[0]` contains the list of numbers, and `mylist[1]` is the list of strings.

ADDING ITEMS TO A LIST

To add items to the end of a list, we use the `append` function. For example, to add a bear burp (I'm sure there is such a thing) to the wizard's shopping list, enter the following:

```
>>> wizard_list.append('bear burp')
>>> print(wizard_list)
['spider legs', 'toe of frog', 'snail tongue', 'slug butter',
'snake dandruff', 'bear burp']
```

You can keep adding more magical items to the wizard's list in the same way, like so:

```
>>> wizard_list.append('mandrake')
>>> wizard_list.append('hemlock')
>>> wizard_list.append('swamp gas')
```

Now the wizard's list looks like this:

```
>>> print(wizard_list)
['spider legs', 'toe of frog', 'snail tongue', 'slug butter',
'snake dandruff', 'bear burp', 'mandrake', 'hemlock', 'swamp gas']
```

The wizard is clearly ready to work some serious magic!

REMOVING ITEMS FROM A LIST

To remove items from a list, use the `del` command (short for *delete*). For example, to remove the fifth item in the wizard's list, snake dandruff, do this:

```
>>> del wizard_list[4]
>>> print(wizard_list)
['spider legs', 'toe of frog', 'snail tongue', 'slug butter',
'bear burp', 'mandrake', 'hemlock', 'swamp gas']
```

NOTE

Remember that positions start at zero, so `wizard_list[4]` *actually refers to the fifth item in the list.*

Try removing the items we just added (mandrake, hemlock, and swamp gas) by entering the following:

```
>>> del wizard_list[7]
>>> del wizard_list[6]
>>> del wizard_list[5]
>>> print(wizard_list)
['spider legs', 'toe of frog', 'snail tongue', 'slug butter',
'bear burp']
```

LIST ARITHMETIC

We can join lists by adding them, just like adding numbers, using a plus (`+`) sign. For example, suppose we have two lists: `list1` , containing the numbers 1 through 4, and `list2` , containing some words. We can add them using the `+` sign, like so:

```
>>> list1 = [1, 2, 3, 4]
>>> list2 = ['I', 'tripped', 'over', 'and', 'hit', 'the', 'floor']
>>> print(list1 + list2)
[1, 2, 3, 4, 'I', 'tripped', 'over', 'and', 'hit', 'the', 'floor']
```

We can also add the two lists and set the result to another variable:

```
>>> list1 = [1, 2, 3, 4]
>>> list2 = ['I', 'ate', 'chocolate', 'and', 'I', 'want', 'more']
>>> list3 = list1 + list2
>>> print(list3)
[1, 2, 3, 4, 'I', 'ate', 'chocolate', 'and', 'I', 'want', 'more']
```

And we can multiply a list by a number. For example, to multiply `list1` by 5, we write `list1 * 5` :

```
>>> list1 = [1, 2]
>>> print(list1 * 5)
[1, 2, 1, 2, 1, 2, 1, 2, 1, 2]
```

This tells Python to repeat `list1` five times, resulting in `1, 2, 1, 2, 1, 2, 1, 2, 1, 2` . On the other hand, division (`/`) and subtraction (`-`) give only errors, as in these examples:

```
>>> list1 / 20
Traceback (most recent call last):
File "<pyshell>", line 1, in <module>
    list1 / 20
TypeError: unsupported operand type(s) for /: 'list' and 'int'

>>> list1 - 20
Traceback (most recent call last):
File "<pyshell>", line 1, in <module>
    list1 - 20
TypeError: unsupported operand type(s) for -: 'list' and 'int'
```

But why? Well, joining lists with `+` and repeating lists with `*` are straightforward enough operations. These concepts also make sense in the real world. For example, if I were to hand you two paper shopping lists and say, "Add these two lists," you might write out all the items on another sheet of paper in order, end to end. The same might be true if I said, "Multiply this list by 3." You could imagine writing a list of all of the list's items three times on another sheet of paper.

But how would you divide a list? For example, consider how you would divide a list of six numbers (1 through 6) in two. Here are just three different ways:

```
[1, 2, 3]     [4, 5, 6]
[1]           [2, 3, 4, 5, 6]
[1, 2, 3, 4]  [5, 6]
```

Would we divide the list in the middle, split it after the first item, or just pick some random place and divide it there? There's no simple answer, and when you ask Python to divide a list, it doesn't know what to do, either. That's why it responds with an error.

For the same reason, you can't add anything other than a list to a list. For example, here's what happens when we try to add the number 50 to `list1` :

```
>>> list1 + 50
Traceback (most recent call last):
  File "<pyshell>", line 1, in <module>
    list1 + 50
TypeError: can only concatenate list (not "int") to list
```

Why do we get an error here? Well, what does it mean to add 50 to a list? Does it mean add 50 to each item? But what if the items aren't numbers? Does it mean add the number 50 to the end or beginning of the list?

In computer programming, commands should work exactly the same way every time you enter them. Your computer sees things only in black and white. Ask it to make a complicated decision, and it throws up its hands with errors.

TUPLES

A *tuple* is like a list that uses parentheses, as in this example:

```
>>> fibs = (0, 1, 1, 2, 3)
>>> print(fibs[3])
2
```

Here we define the variable `fibs` as the numbers 0, 1, 1, 2, and 3. Then, as with a list, we print the item in index position 3 in the tuple by using `print(fibs[3])` .

The main difference between a tuple and a list is that a tuple cannot change once you've created it. For example, if we try to replace the first value in the tuple `fibs` with the number 4 (just as we replaced values in our `wizard_list`), we get an error message:

```
>>> fibs[0] = 4
Traceback (most recent call last):
File "<pyshell>", line 1, in <module>
    fibs[0] = 4
TypeError: 'tuple' object does not support item assignment
```

Why would you use a tuple instead of a list? Well, sometimes it's useful to use something that you know can never change. If you create a tuple with two elements inside, it will always have those two elements.

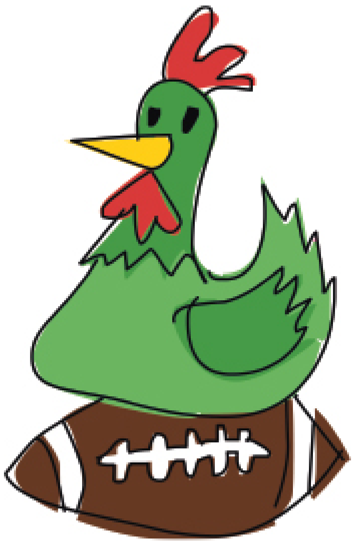
PYTHON DICTIONARIES

In Python, a *dict* (short for *dictionary*) is a collection of things, like lists and tuples. The difference between dicts and lists or tuples is that each item in a dict has a *key* and a corresponding *value* .

For example, say we have a list of people and their favorite sports. We could put this information into a Python list, with the person's name followed by their sport, like so:

```
>>> favorite_sports = ['Ralph Williams, Football',
                       'Michael Tippett, Basketball',
                       'Edward Elgar, Baseball',
                       'Rebecca Clarke, Netball',
                       'Ethel Smyth, Badminton',
                       'Frank Bridge, Rugby']
```

If I asked you what Rebecca Clarke's favorite sport is, you could skim through that list and find the answer is netball. But what if the list included 100 (or many more) people?



If we store this same information in a dictionary, with the person's name as the key and their favorite sport as the value, the code would look like this:

```
>>> favorite_sports = {'Ralph Williams' : 'Football',
                        'Michael Tippett' : 'Basketball',
                        'Edward Elgar' : 'Baseball',
                        'Rebecca Clarke' : 'Netball',
                        'Ethel Smyth' : 'Badminton',
                        'Frank Bridge' : 'Rugby'}
```

We use colons to separate each key from its value, and each key and value is surrounded by single quotes. Notice, too, that the items in a dictionary are enclosed in braces ({}), not parentheses or square brackets. The result is a dict (where each key points to a particular value), as shown in Table 3-1 .

Table 3-1: Keys Pointing to Values in a Dictionary of Favorite Sports

Key	Value
Ralph Williams	Football
Michael Tippett	Basketball

Key	**Value**
Edward Elgar	Baseball
Rebecca Clarke	Netball
Ethel Smyth	Badminton
Frank Bridge	Rugby

Now, to get Rebecca Clarke's favorite sport, we access our dictionary `favorite_sports` by using her name as the key, like so:

```
>>> print(favorite_sports['Rebecca Clarke'])
Netball
```

And the answer is Netball.

To delete a value in a dict, use its key. For example, let's remove Ethel Smyth:

```
>>> del favorite_sports['Ethel Smyth']
>>> print(favorite_sports)
{'Rebecca Clarke': 'Netball', 'Michael Tippett': 'Basketball',
'Ralph Williams': 'Football', 'Edward Elgar': 'Baseball',
'Frank Bridge': 'Rugby'}
```

To replace a value in a dict, we also use its key. Say we need to change Ralph Williams's favorite sport from Football to Ice Hockey. We can do so like this:

```
>>> favorite_sports['Ralph Williams'] = 'Ice Hockey'
>>> print(favorite_sports)
{'Rebecca Clarke': 'Netball', 'Michael Tippett': 'Basketball',
'Ralph Williams': 'Ice Hockey', 'Edward Elgar': 'Baseball',
'Frank Bridge': 'Rugby'}
```

We replace the favorite sport of Football with Ice Hockey by using the key Ralph Williams.

As you can see, working with dictionaries is kind of like working with lists and tuples, except that you can't join dicts with the plus operator (`+`). If you try to do that, you'll get an error message, as in the following example:

```
>>> favorite_sports = {'Rebecca Clarke': 'Netball',
                       'Michael Tippett': 'Basketball',
                       'Ralph Williams': 'Ice Hockey',
                       'Edward Elgar': 'Baseball',
                       'Frank Bridge': 'Rugby'}
>>> favorite_colors = {'Malcolm Warner' : 'Pink polka dots',
                       'James Baxter' : 'Orange stripes',
                       'Sue Lee' : 'Purple paisley'}
>>> favorite_sports + favorite_colors

Traceback (most recent call last):
File "<pyshell>", line 1, in <module>
    favorite_sports + favorite_colors
TypeError: unsupported operand type(s) for +: 'dict' and 'dict'
```

Joining dictionaries doesn't make sense to Python, so it throws up its hands.

WHAT YOU LEARNED

In this chapter, you learned how Python uses strings to store text, and how it uses lists and tuples to handle multiple items. You saw that the items in lists can be changed, and that you can join one list to another list, but that the values in a tuple cannot change. You also learned how to use dictionaries to store values with keys that identify them.

PROGRAMMING PUZZLES

The following are a few experiments you can try yourself. The answers can be found at *http://python-for-kids.com* .

#1: FAVORITES

Make a list of your favorite hobbies and give the list the variable name `games` . Now make a list of your favorite foods and name the variable `foods` . Join the two lists and name the result `favorites` . Lastly, print the variable `favorites` .

#2: COUNTING COMBATANTS

If there are three buildings with 25 ninjas hiding on each roof and two tunnels with 40 samurai hiding inside each tunnel, how many ninjas and samurai are about to do battle? (You can do this with one equation in the Python Shell.)

#3: GREETINGS!

Create two variables: one that points to your first name and one that points to your last name. Now create a string and use placeholders to print your name with a message using those two variables, such as "Hi there, Brando Ickett!"

#4: MULTILINE LETTER

Take the letter we created earlier in the chapter and try to print the exact same text by using a single `print` call (and a multiline string).

4
DRAWING WITH TURTLES

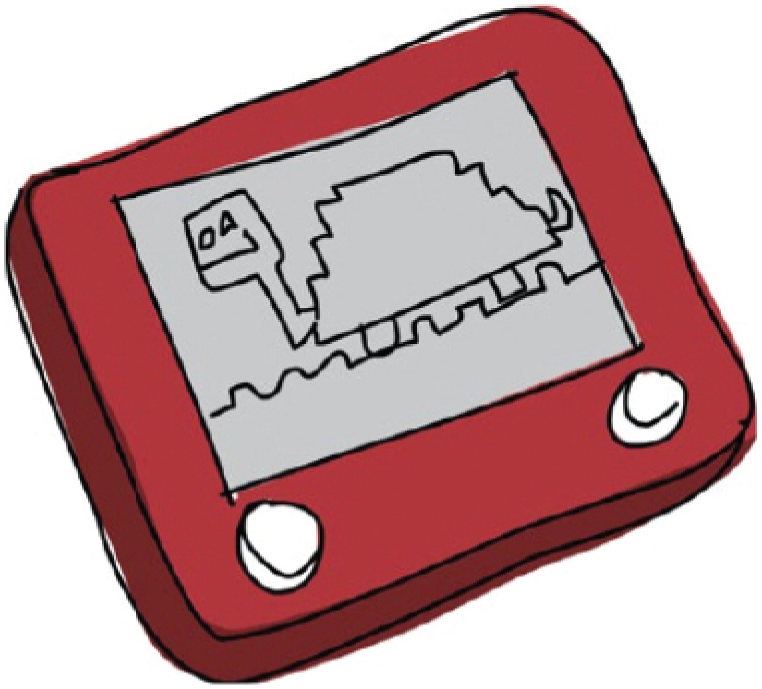
A turtle in Python is not quite like a turtle in the real world. We know a turtle as a reptile that moves around very slowly and carries its house on its back. In the world of Python, a *turtle* is a small, black arrow that moves slowly around the screen. Actually, considering that a Python turtle leaves a trail as it moves around the screen, it's less like a turtle and more like a snail or slug.

In this chapter, we'll use a Python turtle to learn the basics of computer graphics by drawing some simple shapes and lines.

USING PYTHON'S TURTLE MODULE

A *module* in Python is a way for programmers to make useful code available for other programmers to use. (Among other things, a module can contain *functions* we can use.) We'll learn more about modules and functions in Chapter 7 .

The turtle is a special module in Python that we can use to learn how computers draw pictures on a screen. The `turtle` module is a way of programming vector graphics, which is basically just drawing with simple lines, dots, and curves.



Let's see how the turtle works. First, start the Python Shell. Next, tell Python to use the turtle by importing the `turtle` module, as follows:

```
>>> import turtle
```

Importing a module tells Python that you want to use it.

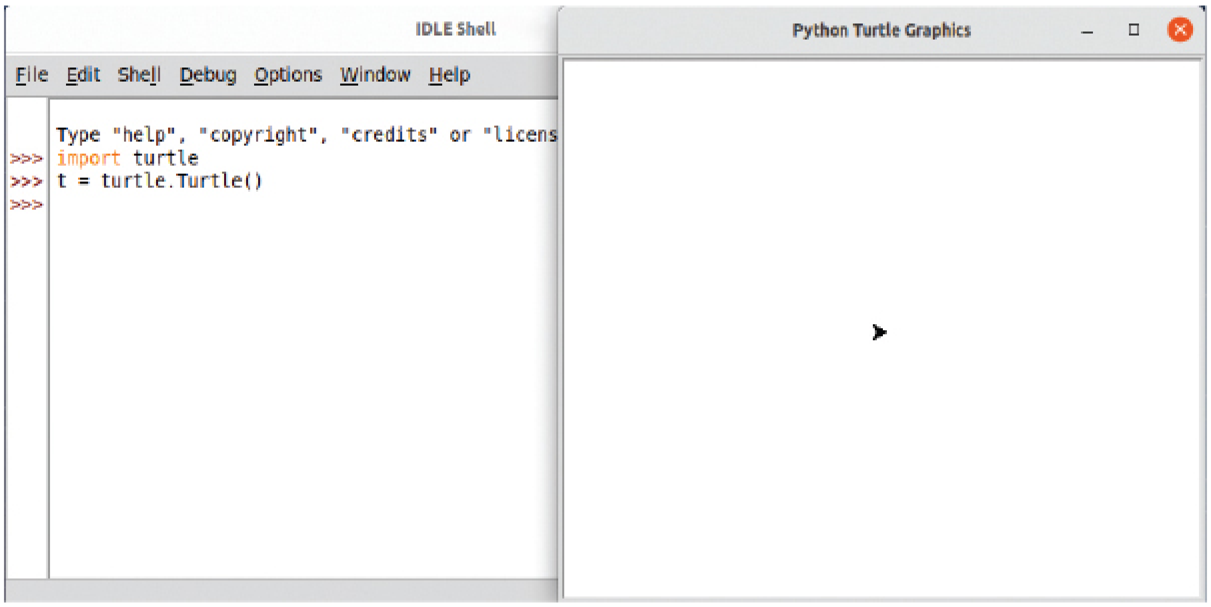
CREATING A CANVAS

Now that we've imported the `turtle` module, we need to create a *canvas* —a blank space to draw on, like an artist's canvas. To do so, we call the function `Turtle` from the `turtle` module, which automatically creates a canvas (we'll learn more about what a function is in Chapter 7).

Enter this into the Python Shell:

```
>>> t = turtle.Turtle()
```

You should see a blank box (the canvas), with an arrow in the center (similar to Figure 4-1). The arrow in the middle of the screen is the turtle, and you're right—it isn't very turtle-like.



```
Type "help", "copyright", "credits" or "licens
>>> import turtle
>>> t = turtle.Turtle()
>>>
```

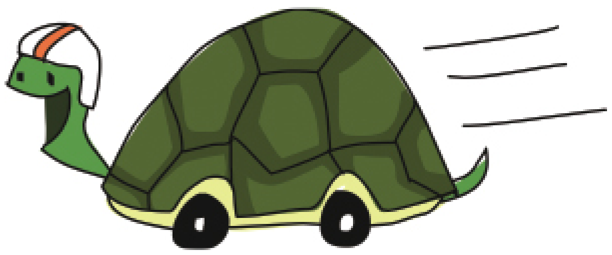
Figure 4-1: Running turtle in the Python Shell

NOTE

If the turtle module doesn't appear to be working, there are some additional steps you can try in Appendix C .

MOVING THE TURTLE

You can send instructions to the turtle by using functions available on the variable `t` we just created, similar to using the `Turtle` function in the `turtle` module. For example, the `forward` instruction tells the turtle to move forward. To tell the turtle to advance 50 pixels, enter the following command:



```
>>> t.forward(50)
```

You should see something like Figure 4-2 .

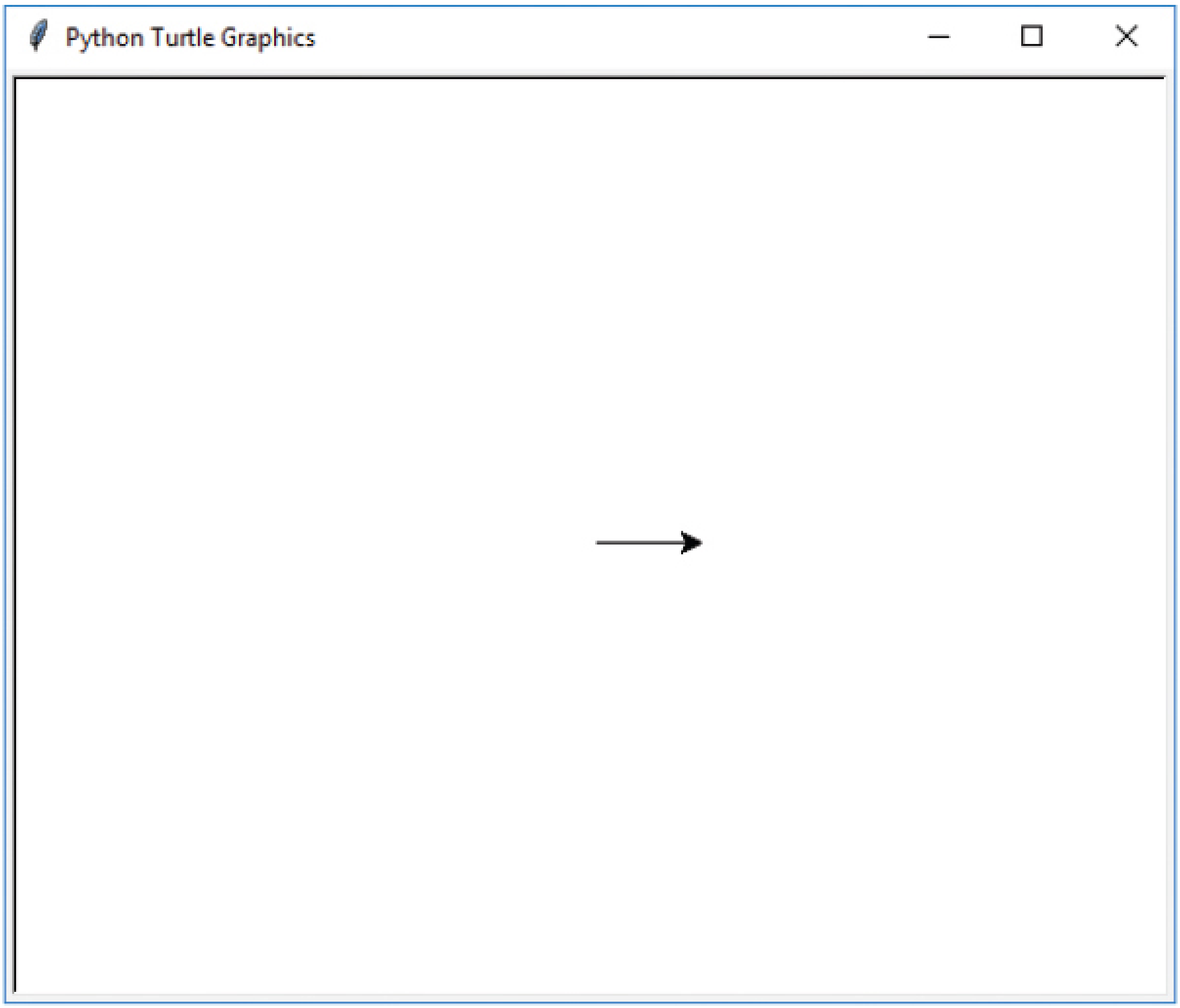


Figure 4-2: The turtle moving forward

The turtle has moved forward 50 pixels. A *pixel* is a single point on the screen—the smallest element that can be represented. Everything you see on your computer monitor is made up of pixels, which are tiny, square dots. If you could zoom in on the canvas and the line drawn by the turtle, you would be able to see that the arrow representing the turtle's path is just a bunch of pixels. This is the basis for simple computer graphics.

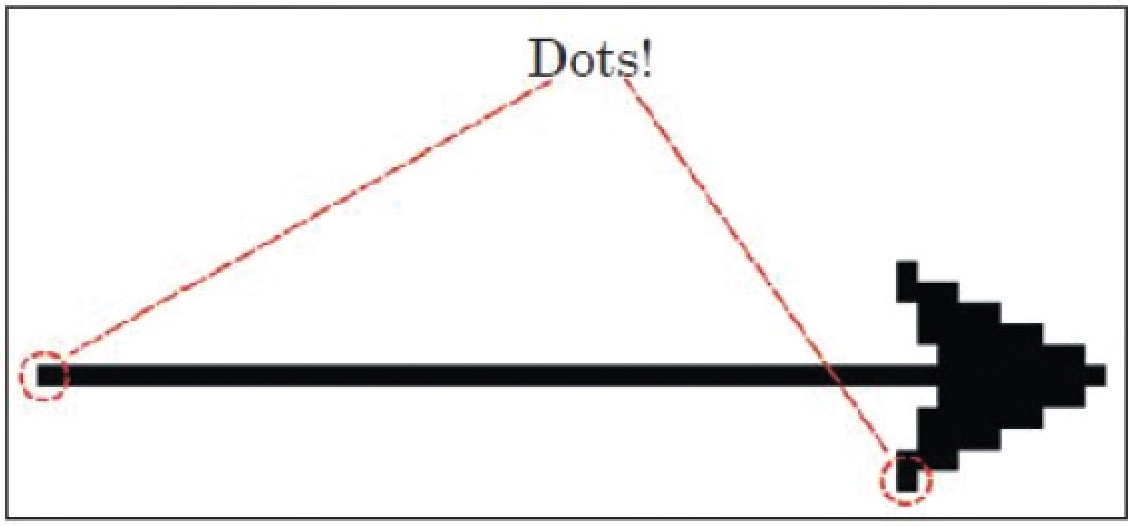


Figure 4-3: Pixels are dots!

Now we'll tell the turtle to turn 90 degrees to the left with the following command:

```
>>> t.left(90)
```

If you haven't learned about *degrees* yet, imagine you're standing in the center of a circle:

- The direction you're facing is 0 degrees.
- If you hold out your left arm, that's 90 degrees left.
- If you hold out your right arm, that's 90 degrees right.

You can see this 90-degree turn to the left or right in Figure 4-4 .

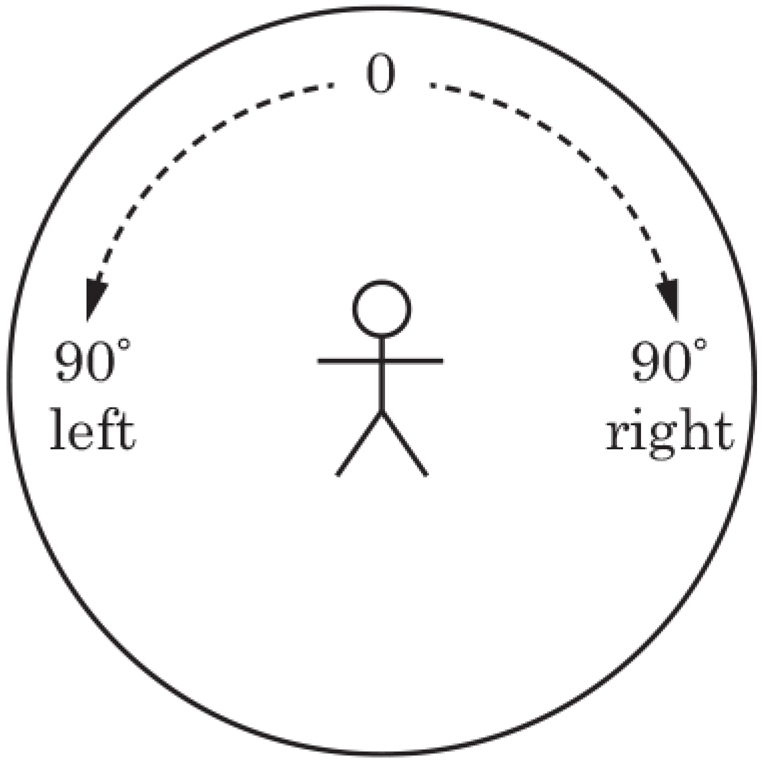


Figure 4-4: 90 degrees left and right

If you continue around the circle to the right from where your right arm is pointing, 180 degrees is directly behind you, 270 degrees is the direction your left arm is pointing, and 360 degrees is back where you started; degrees go from 0 to 360. The degrees in a full circle, when turning to the right, can be seen in 45-degree increments in Figure 4-5 .

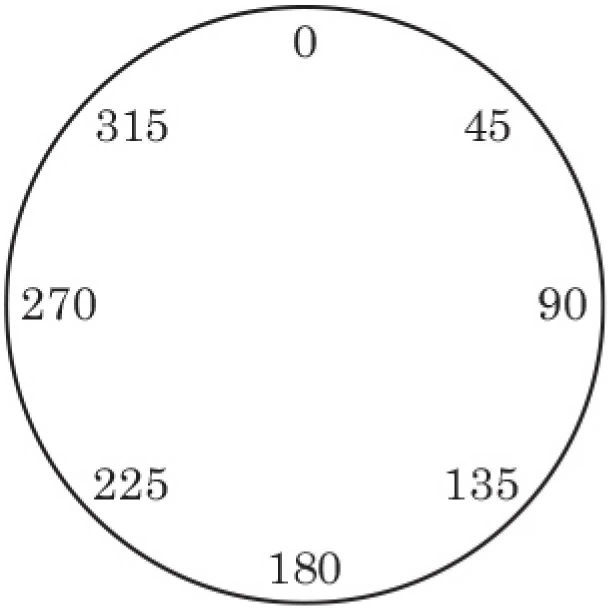


Figure 4-5: 45-degree increments

When Python's turtle turns left, it swivels around to face the new direction (just as if you turned your body to face where your arm is pointing 90 degrees left). The `t.left(90)` command points the arrow up (since it started by pointing to the right). You can see this in Figure 4-6 .

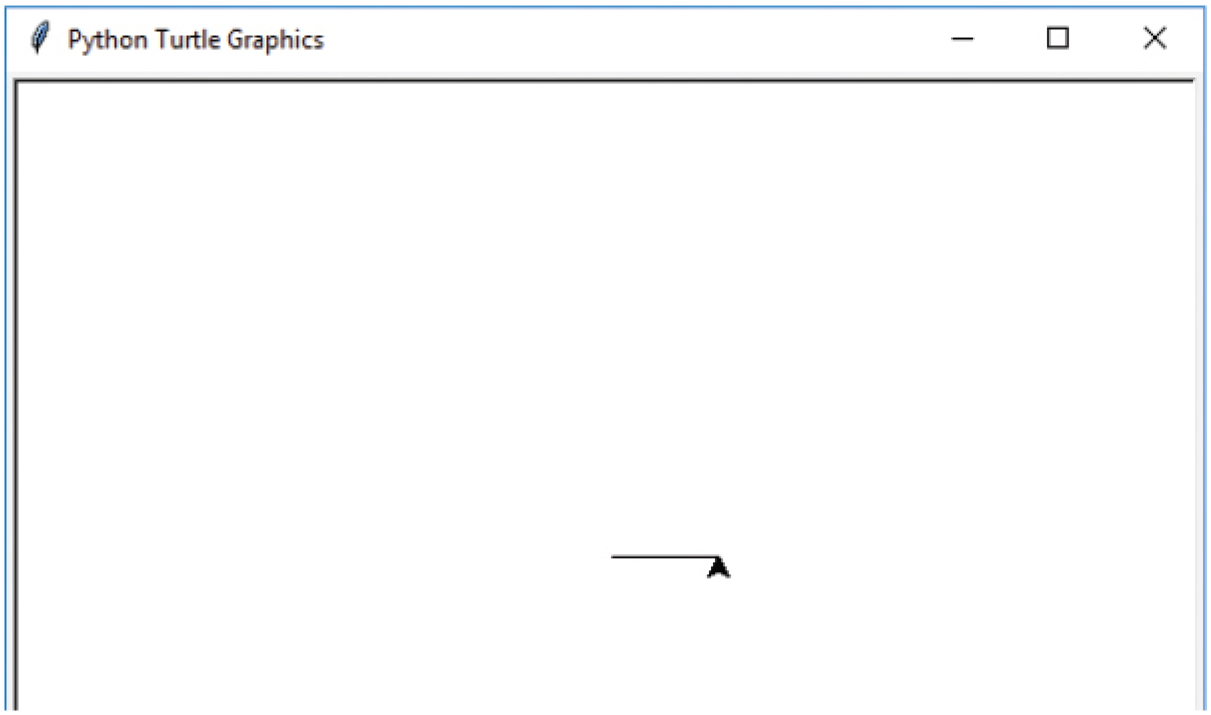


Figure 4-6: Turtle after turning left

NOTE

When you call `t.left(90)` *, it's the same as calling* `t.right(270)` *in terms of the direction the turtle ends up facing at the end. This is also true when calling* `t.right(90)` *, which is the same as* `t.left(270)` *. Just imagine that circle and follow along with the degrees.*

Now we'll draw a square. Add the following code to the lines you've already entered:

```
>>> t.forward(50)
>>> t.left(90)
>>> t.forward(50)
>>> t.left(90)
>>> t.forward(50)
>>> t.left(90)
```

Your turtle should have drawn a square and should now be facing in the same direction it started, as seen in Figure 4-7 .

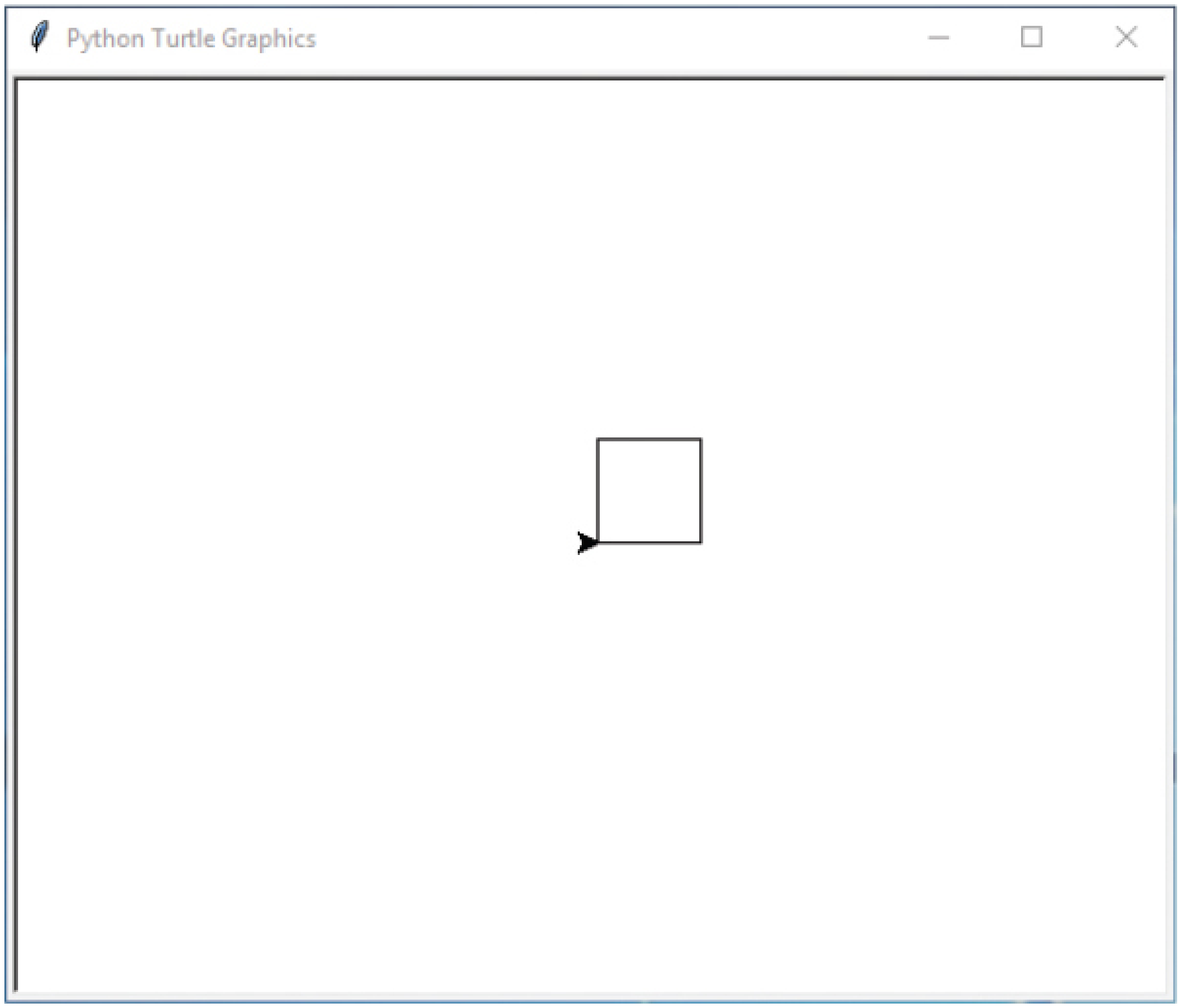


Figure 4-7: Turtle drawing a square

To erase the canvas, enter `t.reset()` . This clears the canvas and puts the turtle back at its starting position.

```
>>> t.reset()
```

You can also use `t.clear()` , which just clears the screen and leaves the turtle where it is.

```
>>> t.clear()
```

We can also turn our turtle right or move it backward. We can use `up` to lift the "pen" off the page (in other words, tell the turtle to stop drawing), and `down` to start drawing again. These functions are written in the same way as the others we've used.

Let's try another drawing using some of these commands. This time, we'll have the turtle draw two lines. Enter the following code:

```
>>> t.reset()
>>> t.backward(100)
>>> t.up()
>>> t.right(90)
>>> t.forward(20)
>>> t.left(90)
>>> t.down()
>>> t.forward(100)
```

First, we clear the canvas and move the turtle back to its starting position with `t.reset()` . Next, we move the turtle backward 100 pixels with `t.backward(100)` , and then use `t.up()` to pick up the pen and stop drawing.

Then, with the `t.right(90)` command, we turn the turtle right 90 degrees to point down, toward the bottom of the screen, and with `t.forward(20)` , we move forward 20 pixels. Nothing is drawn because we used the `up` command on the third line. Next, we turn the turtle left 90 degrees to face right with `t.left(90)` , and then we use the `down` command to tell the turtle to start drawing again. Lastly, we draw a line forward, parallel to the first line we drew, with `t.forward(100)` . The two parallel lines we've drawn end up looking like Figure 4-8 .

Wooo!!

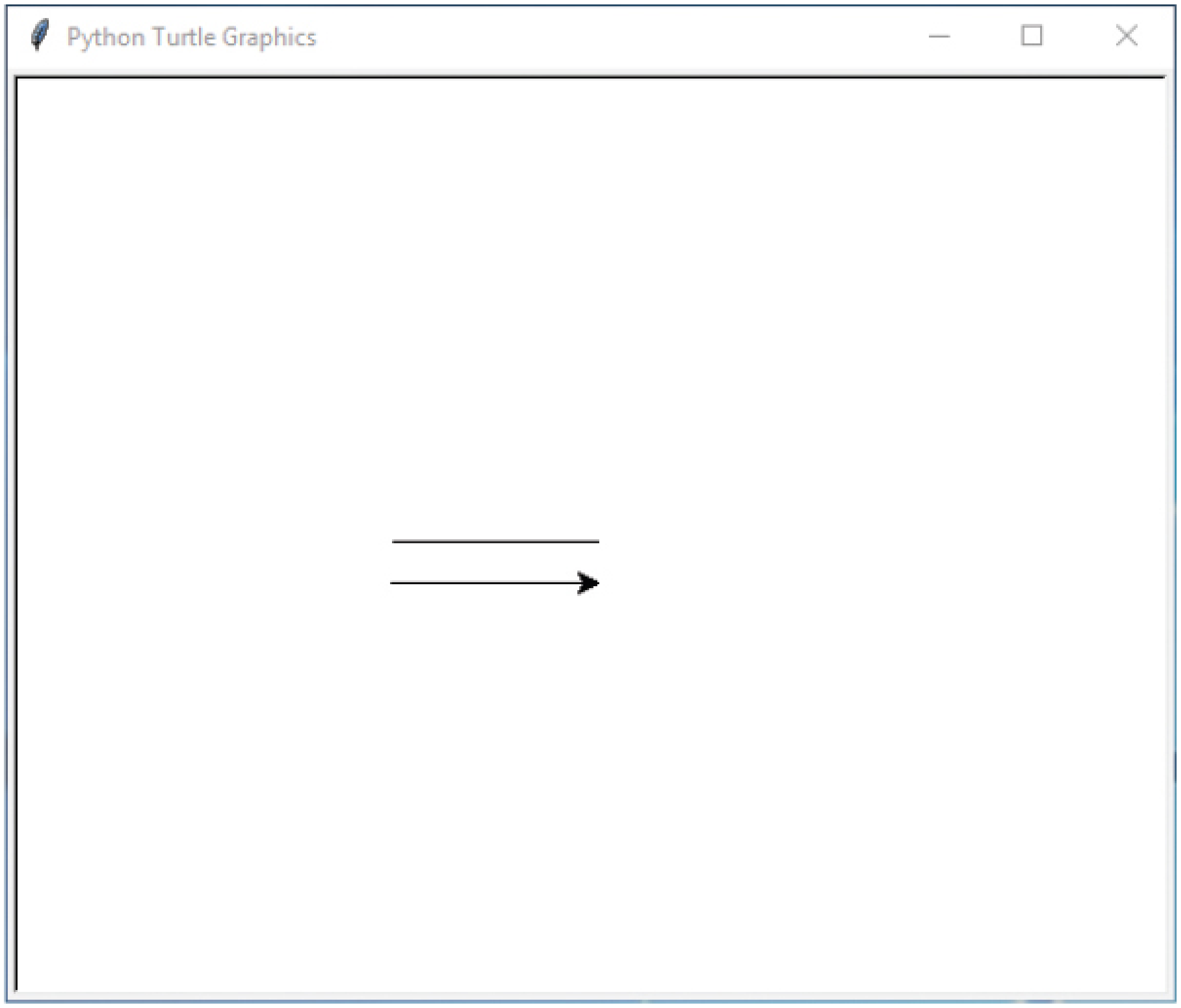

Python Turtle Graphics

Figure 4-8: Turtle drawing parallel lines

WHAT YOU LEARNED

In this chapter, you learned how to use Python's `turtle` module. We drew some simple lines, using left and right turns and forward and backward commands. You found out how to stop the turtle from drawing by using `up` , and start drawing again with the `down` command. You also discovered that the turtle turns by degrees.

PROGRAMMING PUZZLES

Try drawing some of the following shapes with the turtle. The solutions can be found at *http://python-for-kids.com* .

#1: A RECTANGLE

Create a new canvas using the `turtle` module's `Turtle` function and then draw a rectangle.

#2: A TRIANGLE

Create another canvas and draw a triangle. Look back at the diagram of the circle with the degrees ("Moving the Turtle" on page 45) to remind yourself which direction to turn the turtle.

#3: A BOX WITHOUT CORNERS

Write a program to draw the four lines shown in Figure 4-9 (the size isn't important, just the shape).

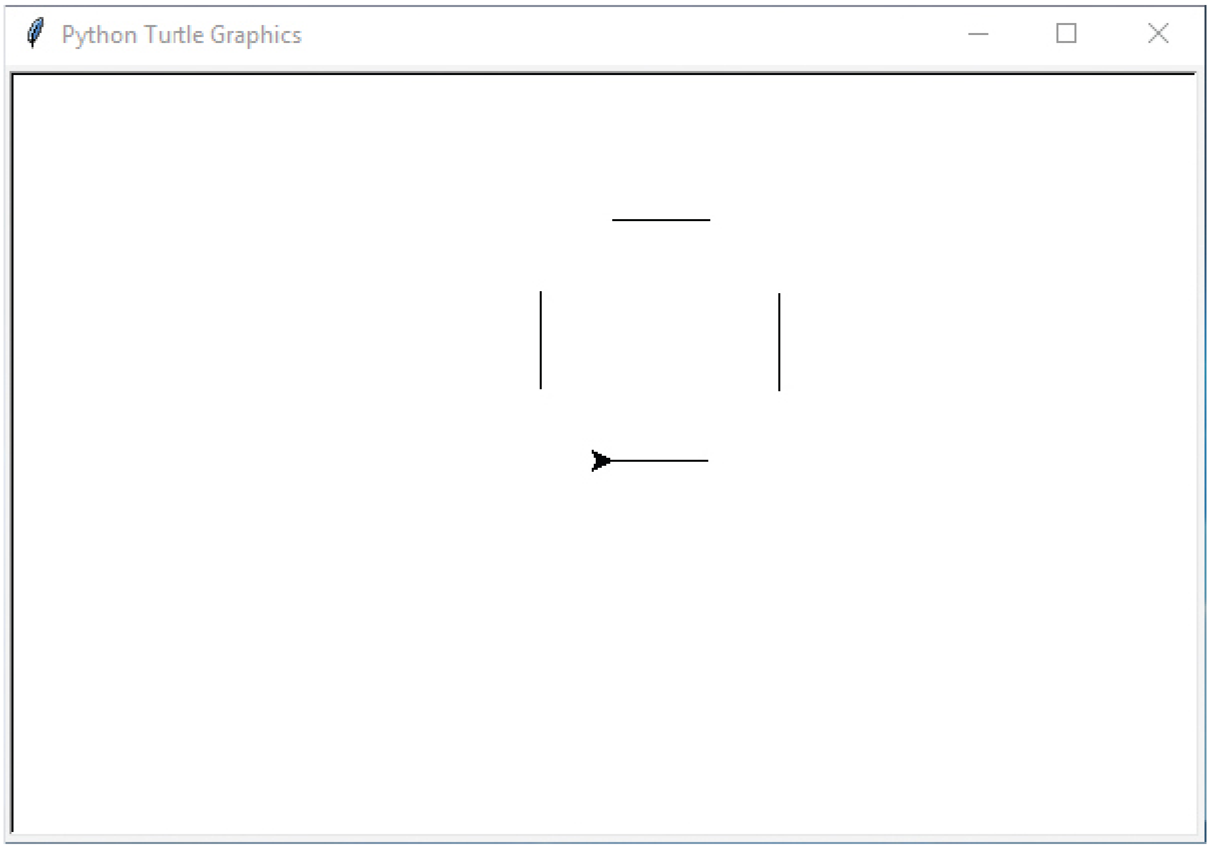


Figure 4-9: A box without corners in Turtle

#4: A TILTED BOX WITHOUT CORNERS

Write a program to draw the four lines shown in Figure 4-10 (similar to the previous puzzle, but the box is tilted on its side). Again, the size of the box isn't important—just the shape.

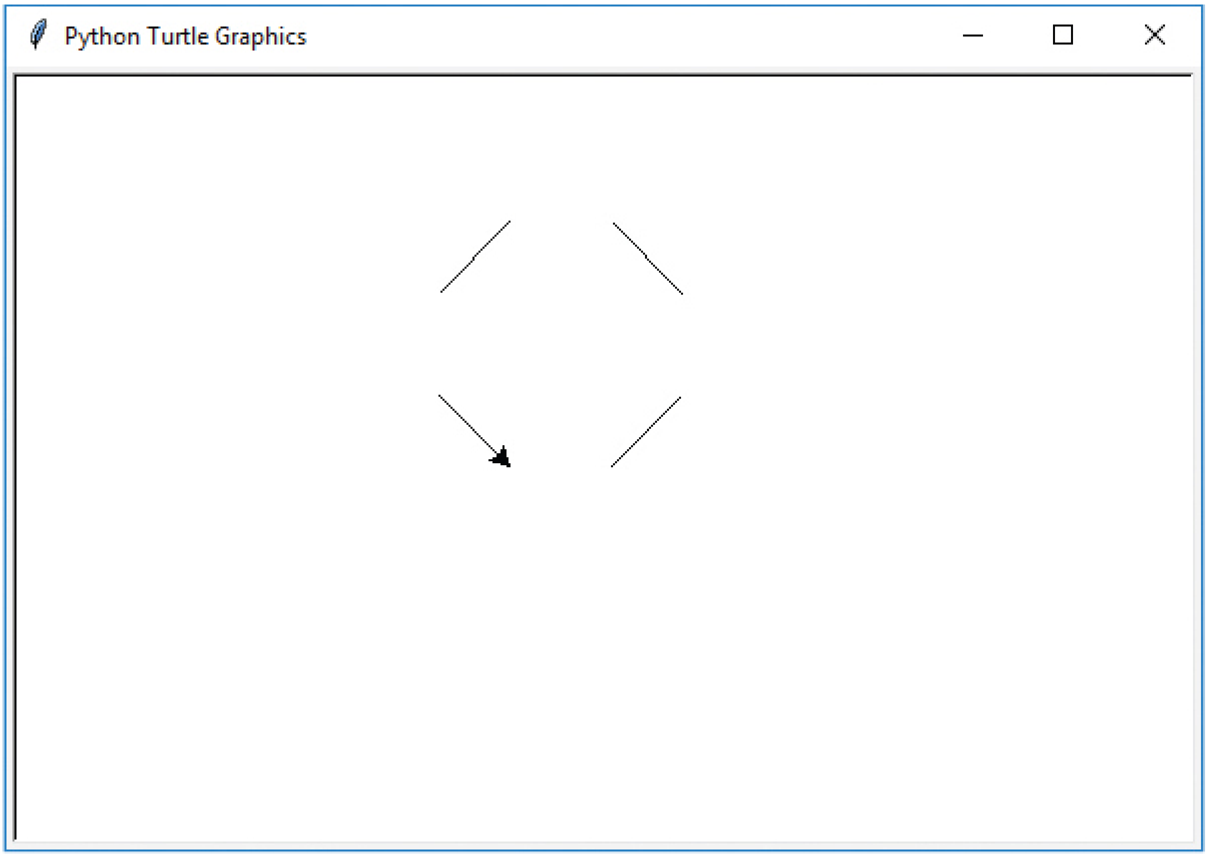


Figure 4-10: A tilted box without corners in Turtle

5
ASKING QUESTIONS WITH IF AND ELSE

In programming, we often ask yes or no questions, and do something based on the answer. For example, we might ask, “Are you older than 20?” and, if the answer is yes, respond with “You are too old!” These sorts of questions are called *conditions* , and we combine conditions and their responses into `if` statements. Conditions can be more complicated than a single question, and `if` statements can be combined with multiple questions and different responses based on the answer to each question. In this chapter, you’ll learn to use `if` statements to build programs.

IF STATEMENTS

We might write an `if` statement in Python like this:

```
>>> age = 13
>>> if age > 20:
        print('You are too old!')
```

An `if` statement is made up of the `if` keyword, followed by a condition and a colon (`:`), as in this `if age > 20:` statement. The lines following the colon must be in a block; if the answer to the question is yes (or `True`), the commands in the block will run. Now, let's explore how to write blocks and conditions.

NOTE

`True` is a Boolean *value, named after mathematician George Boole. Booleans can only have one of two values: `True` or `False`.*

A BLOCK IS A GROUP OF PROGRAMMING STATEMENTS

A *block* of code is a grouped set of programming statements. For example, when `if age > 20:` is `True`, you might want to do more than just print "You are too old!" Perhaps you want to print out more sentences, like this:

```
>>> age = 25
>>> if age > 20:
        print('You are too old!')
        print('Why are you here?')
        print('Why aren\'t you mowing a lawn or sorting papers?')
```

This block of code is made up of three `print` calls that are run only if the condition `age > 20` is found to be `True`. Each line in the block has four spaces at the beginning. Let's look at that code again, with visible spaces:

```
>>> age = 25
>>> if age > 20:
 .... print('You are too old!')
 .... print('Why are you here?')
 .... print('Why aren\'t you mowing a lawn or sorting papers?')
```

In Python, *whitespace* —such as a tab (inserted when you press the TAB key) or a space (inserted when you press the spacebar)—is meaningful. Code that is at the same position (or indented the same number of spaces from the left margin) is grouped into a block. Whenever you start a new line with more spaces than the previous one, you are starting a new block. This new block is also part of the previous block, like Figure 5-1 .

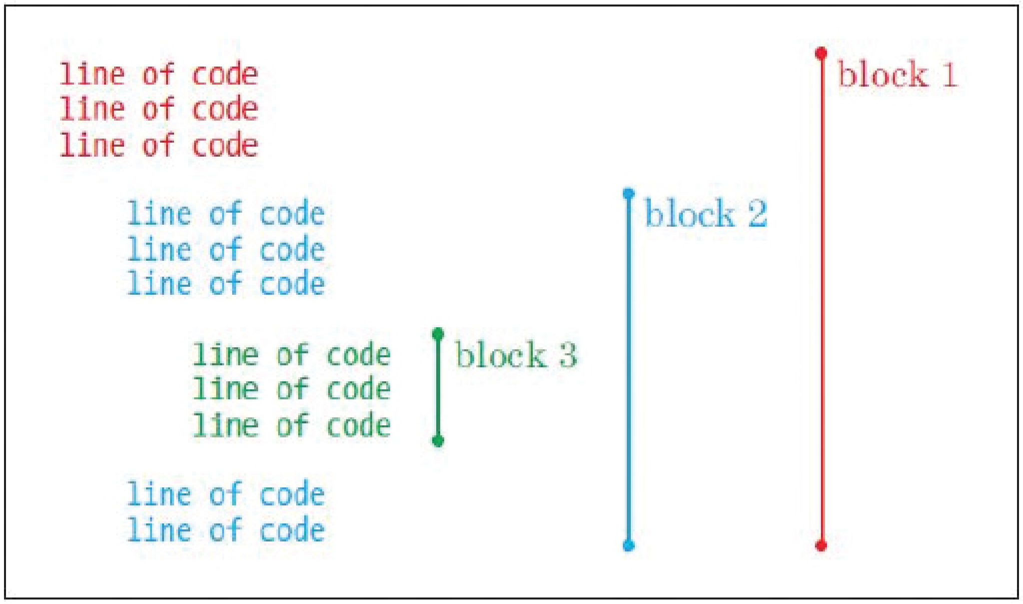


Figure 5-1: How blocks of code work

We group statements into blocks because they are related and need to be run together. When you change the indentation of code, you're generally creating new blocks. Figure 5-2 shows how three blocks are created just by changing the indentation.

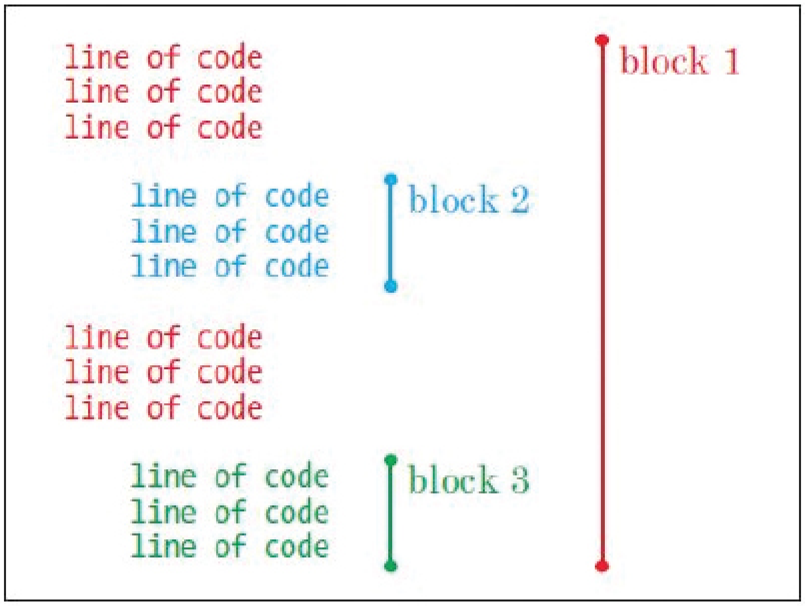


Figure 5-2: A second example showing how blocks of code work

Here, even though blocks 2 and 3 have the same indentation, they are considered different blocks because there is a block with less indentation (fewer spaces) between them.

A block with four spaces on one line and six spaces on the next will produce an *indentation error* when you run it. This is because Python expects you to use the same number of spaces for all lines in a block. Here's an example:

```
>>> if age > 20:
 .... print('You are too old!')
 ...... print('Why are you here?')
```

I've made the spaces visible so you can see the differences. Notice that the second `print` line has six spaces instead of four. When we try to run this code, Python highlights the problem line with a red block and displays an explanatory `SyntaxError` message:

```
>>> age = 25
>>> if age > 20:
        print('You are too old!')
          print('Why are you here?')
SyntaxError: unexpected indent
```

Python didn't expect to see two extra spaces at the beginning of the second `print` line.

NOTE

Use consistent spacing to make your code easier to read. If you start writing a program and put four spaces at the beginning of a block, keep using four spaces at the beginning of the other blocks in your program. Be sure to indent each line in one block with the same number of spaces.

CONDITIONS HELP US COMPARE THINGS

A *condition* is a programming expression that compares things and tells us whether the criteria set by the comparison are `True` (yes) or `False` (no). For example, `age > 10` is a condition that asks, "Is the value of the `age` variable greater than 10?" Another condition might be `hair_color == 'mauve'` , or, "Is the value of the `hair_color` variable mauve?"

We use symbols in Python—called *operators* —to create our conditions, such as *equal to* , *greater than* , and *less than* . Table 5-1 lists common operators.

Table 5-1: Symbols for Conditions

Symbol	Definition
`==`	Equal to
`!=`	Not equal to
`>`	Greater than
`<`	Less than
`>=`	Greater than or equal to
`<=`	Less than or equal to

For example, if you are 10 years old, the condition `your_age == 10` would return `True` ; otherwise, it would return `False` . If you are 12 years old, the condition `your_age > 10` would return `True` .

NOTE

Be sure to use a double equal sign (`==`) when defining an equal-to condition.

Let's try some examples. Here, we set our age as equal to 10 and then write a conditional statement that will print "You are too old for my jokes!" if `age` is greater than 10:

```
>>> age = 10
>>> if age > 10:
        print('You are too old for my jokes!')
```

What happens when we type this into the Python Shell and press ENTER?

Nothing.

Because the value returned by `age` is not greater than 10, Python does not run the `print` block. However, if we had set `age` to 20, the message would print.

Now let's change the previous example to use a greater-than-or-equal-to (`>=`) condition:

```
>>> age = 10
>>> if age >= 10:
        print('You are too old for my jokes!')
```

You should see "You are too old for my jokes!" printed to the screen because the value of `age` is equal to 10.

Next, let's try using an equal-to (`==`) condition:

```
>>> age = 10
>>> if age == 10:
        print("What's brown and sticky? A stick!!")
```

You should see the message "What's brown and sticky? A stick!!" printed to the screen.

IF-THEN-ELSE STATEMENTS

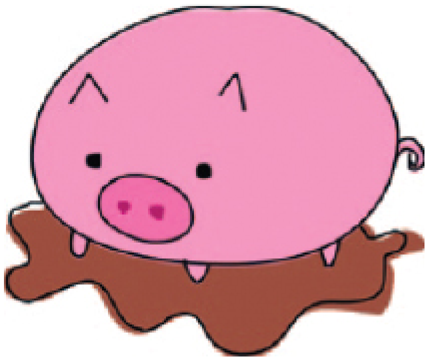
In addition to using `if` statements to do something when a condition is met (`True`), we can also use `if` statements to do something when a condition is not true. For example, we might print one message to the screen if your age is 12 and another if it's not 12.

The trick here is to use an if-then-else statement, which essentially says, "If something is true, then do this; else, do that."

Let's create an if-then-else statement. Enter the following into the Python Shell:

```
>>> print('Want to hear a dirty joke?')
Want to hear a dirty joke?
>>> age = 12
>>> if age == 12:
        print('A pig fell in the mud!')
    else:
        print("Shh. It's a secret.")

A pig fell in the mud!
```

Because we've set `age` to 12, and the condition is asking whether `age` is equal to 12, you should see the first `print` message on the screen. Now try changing the value of `age` to a number other than 12, like this:



```
>>> print('Want to hear a dirty joke?')
Want to hear a dirty joke?
>>> age = 8
>>> if age == 12:
        print('A pig fell in the mud!')
    else:
        print("Shh. It's a secret.")

Shh. It's a secret.
```

This time, you should see the second `print` message.

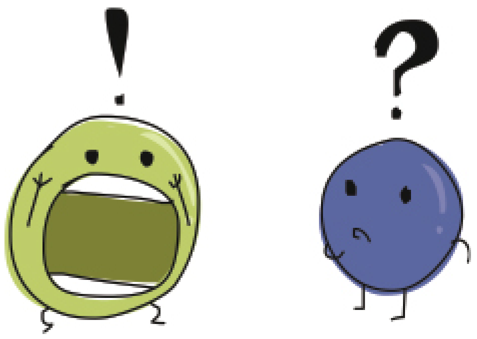
IF AND ELIF STATEMENTS

We can extend an `if` statement even further with `elif` , which is short for *else-if* . These statements differ from if-then-else statements in that there can be more than one `elif` in the same statement. For example, we can check if a person's age is 10, 11, or 12 (and so on) and have our program do something different based on the answer:

```
>>> age = 12
>>> if age == 10:
        print('What do you call an unhappy cranberry?')
        print('A blueberry!')
    elif age == 11:
        print('What did the green grape say to the blue grape?')
        print('Breathe! Breathe!')
    elif age == 12:
        print('What did 0 say to 8?')
        print('Hi guys!')
    elif age == 13:
        print("Why wasn't 10 afraid of 7?")
        print('Because rather than eating 9, 7 8 pi.')
    else:
        print('Huh?')

What did 0 say to 8?
Hi guys!
```

In this example, the `if` statement on the second line checks if the value of `age` is equal to 10. If so, the `print` function that follows is run. However, because we've set `age` equal to 12, the computer jumps to the next part of the `if` statement (the first `elif` or *else if*) and checks if the value of `age` is equal to 11. It isn't, so the computer jumps to the next `elif` to see if `age` is equal to 12. It is, so this time the computer executes the following `print` call.

When you enter this code in IDLE, it will automatically indent, so be sure to press the BACKSPACE key (or DELETE key if you're using a Mac) once you've typed each `print` statement. That way, your `if` , `elif` , and `else` statements will start at the far-left margin. This is the same position the `if` statement would be in, if the prompt (`>>>`) were absent.



COMBINING CONDITIONS

You can combine conditions using the keywords `and` and `or` , which produces shorter and simpler code. Here's an example using `or` :

```
>>> if age == 10 or age == 11 or age == 12 or age == 13:
        print('What is 13 + 49 + 84 + 155 + 97? A headache!')
    else:
        print('Huh?')
```

In this code, if any of the conditions on the first line are true (if `age` is 10, 11, 12, or 13), the `print` statement on the following line will run.

If the conditions in the first line are not true (`else`), Python displays `Huh?` on the screen.

To shrink this example even further, we could use the `and` keyword, along with the greater-than-or-equal-to operator (`>=`) and less-than-or-equal-to operator (`<=`), as follows:

```
>>> if age >= 10 and age <= 13:
        print('What is 13 + 49 + 84 + 155 + 97? A headache!')
    else:
        print('Huh?')
```

Here, if `age` is greater than or equal to 10 and less than or equal to 13, the `print` statement on the following line will run. For example, if the value of `age` is 12, then `What is 13 + 49 + 84 + 155 + 97? A headache!` will be printed to the screen because 12 is more than 10 and less than 13.

VARIABLES WITH NO VALUE—NONE

Just as we can assign numbers, strings, and lists to a variable, we can also assign nothing, or an empty value. In Python, an empty value is referred to as `None` . It's important to note that the value `None` is different from the value `0` because it is the *absence* of a value rather than a number with a value of 0. Here's an example where we set a variable to `None` :

```
>>> myval = None
>>> print(myval)
None
```

Assigning `None` to a variable tells Python that the variable no longer has any value (or rather, that it no longer labels a value). It's also a way to define a variable without setting its value. This might be useful when you know you'll need a variable later in your program but you want to define all variables at the beginning.

NOTE

Programmers often define their variables at the beginning of a program (or a function) to have a quick reference of all the variables used in a chunk of code.

You can check for `None` in an `if` statement as well, as in the following example:

```
>>> myval = None
>>> if myval is None:
        print("The variable myval doesn't have a value")

The variable myval doesn't have a value
```

This is useful when you want to calculate a value for a variable only if it hasn't already been calculated yet. In this scenario, checking for `None` tells Python to do the calculation only if the variable doesn't have a value.

THE DIFFERENCE BETWEEN STRINGS AND NUMBERS

User input is what a person enters on the keyboard—whether that's a character, a pressed arrow or ENTER key, or anything else. In Python, user input is a string, meaning when you type the number 10 on your keyboard, Python saves the number 10 as a string, not a number.

Compare the number 10 and the string `'10'` . While we see the only difference between the two being that one is surrounded by quotes, to a computer, they are completely different.

For example, let's compare the value of the variable `age` to a number in an `if` statement, like so:

```
>>> if age == 10:
        print("What's the best way to speak to a monster?")
        print("From as far away as possible!")
```

If we set the variable `age` to the number `10` first:

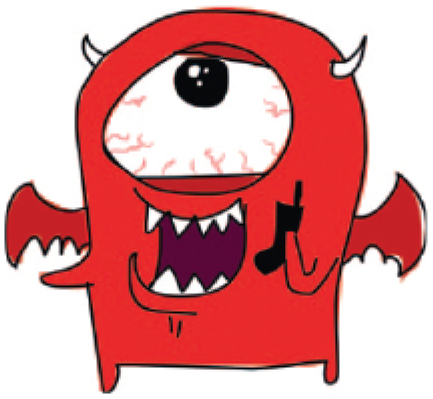
```
>>> age = 10
>>> if age == 10:
        print("What's the best way to speak to a monster?")
        print("From as far away as possible!")

What's the best way to speak to a monster?
From as far away as possible!
```

As you can see, the `print` statement executes.

Next, we set `age` to the string `'10'` (with quotes), like this:

```
>>> age = '10'
>>> if age == 10:
        print("What's the best way to speak to a monster?")
        print("From as far away as possible!")

>>>
```

Here, the `print` statement doesn't run because Python doesn't see the string as a number.



Fortunately, Python has functions that can turn strings into numbers and numbers into strings. For example, you can convert the string `'10'` into a number with the `int` function:

```
>>> age = '10'
>>> converted_age = int(age)
```

The variable `converted_age` now holds the number 10 (rather than a string).

To convert a number into a string, use `str` like so:

```
>>> age = 10
>>> converted_age = str(age)
```

Now `converted_age` holds the string `'10'` instead of the number 10.

Remember the `if age == 10` statement that didn't print anything when the variable was set to a string (`age = '10'`)? If we convert the variable first, we get an entirely different result:

```
>>> age = '10'
>>> converted_age = int(age)
>>> if converted_age == 10:
        print("What's the best way to speak to a monster?")
        print("From as far away as possible!")

What's the best way to speak to a monster?
From as far away as possible!
```

But know this: if you try to convert a number with a decimal point (also called *floating point numbers* , because the dot can "move" around in a number), you'll get an error because the `int` function expects an *integer* (a number without a decimal place):

```
>>> age = '10.5'
>>> converted_age = int(age)
Traceback (most recent call last):
    File "<pyshell#35>", line 1, in <module>
    converted_age = int(age)
ValueError: invalid literal for int() with base 10: '10.5'
```

Python sends a `ValueError` to tell you that the value you've tried to use isn't appropriate. To fix this, use `float` instead of `int` , as the `float` function can handle numbers that aren’t integers:

```
>>> age = '10.5'
>>> converted_age = float(age)
>>> print(converted_age)
10.5
```

You’ll also get a `ValueError` if you try to convert a string that doesn’t contain a number in digits:

```
>>> age = 'ten'
>>> converted_age = int(age)
Traceback (most recent call last):
    File "<pyshell#1>", line 1, in <module>
    converted_age = int(age)}
ValueError: invalid literal for int() with base 10: 'ten'
```

Because we used the word `ten` rather than the number `10` , Python throws up its hands.

WHAT YOU LEARNED

In this chapter, you learned to use `if` statements to create blocks of code that are executed only when particular conditions are true. You saw how to extend `if` statements by using `elif` so that different sections of code will execute as a result of different conditions, and how to use the `else` keyword to execute code if none of the conditions turn out to be true.

You combined conditions by using the `and` and `or` keywords to check if numbers fall in a range, and changed strings into numbers with the `int` , `str` , and `float` functions. You also discovered that `None` can reset variables to their initial, empty state.

PROGRAMMING PUZZLES

Try the following puzzles using `if` and conditions. The solutions can be downloaded at *http://python-for-kids.com* .

#1: ARE YOU RICH?

What do you think the following code will do? Try to figure out the answer without typing it into the Python Shell and then check your work.

```
>>> money = 2000
>>> if money > 1000:
        print("I'm rich!!")
    else:
        print("I'm not rich!!")
          print("But I might be later...")
```

#2: TWINKIES!

Create an `if` statement that checks whether a number of Twink-ies (in the variable `twinkies`) is less than 100 or greater than 500. Your program should print the message “Too few or too many” if the condition is `True` .

#3: JUST THE RIGHT NUMBER

Create an `if` statement that checks whether the amount of money contained in the `money` variable is between 100 and 500 or between 1,000 and 5,000.

#4: I CAN FIGHT THOSE NINJAS

Create an `if` statement that prints “That’s too many” if the variable `ninjas` contains a number less than 50; prints “It’ll be a struggle, but I can take ’em” if it’s less than 30; and prints “I can fight those ninjas!” if it’s less than 10. You might try out your code with this:

```
>>> ninjas = 5
```

6
GOING LOOPY

Nothing is worse than needing to do the same thing over and over. There's a reason we're told to count sheep when we're having trouble falling asleep, and it has nothing to do with the amazing sleep-inducing powers of woolly mammals. It's because endless repetition is boring, and your mind can drift off to sleep easily if you're not focusing on something interesting.

Programmers don't like repeating themselves either, unless they're also trying to fall asleep. Thankfully, most programming languages have a `for` loop, which repeats things like statements and blocks of code automatically.

In this chapter, we'll look at `for` loops, as well as another type of loop that Python offers: the `while` loop.

USING FOR LOOPS

To print `hello` five times in Python, you *could* do the following:

```
>>> print('hello')
hello
>>> print('hello')
hello
>>> print('hello')
hello
>>> print('hello')
hello
>>> print('hello')
hello
```

But this is rather tedious. Instead, you can use a `for` loop to reduce the amount of typing and repetition, like this:

```
❶ >>> for x in range(0, 5):
       ❷ print('hello')

  hello
  hello
  hello
  hello
  hello
```

The `range` function ❶ can create a list of numbers ranging from a starting number up to the number right before the ending number.

This may seem a little confusing, so let's combine the `range` function with the `list` function to see exactly how this works. The `range` function doesn't actually create a list of numbers; it returns an *iterator* , which is a Python object designed to work with loops. However, if we combine `range` with `list` , we get a list of numbers:

```
>>> print(list(range(10, 20)))
[10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
```

In our previous example, `for x in range(0,5):` is actually telling Python to do the following:

1. Start counting from 0 and stop before reaching 5.
2. For each number we count, store the value in the `x` variable.

Then, Python executes the `print('hello')` statement. Note the four additional spaces at the beginning of the line ❷ (as compared to the first line ❶). IDLE should have automatically indented this for you. When we hit ENTER after the second line, Python prints `hello` five times.

We could also use the variable `x` in our `print` statement to count the number of hellos:

```
>>> for x in range(0, 5):
        print(f'hello {x}')
hello 0
hello 1
hello 2
hello 3
hello 4
```

If we get rid of the `for` loop, our code might look something like this:

```
>>> x = 0
>>> print(f'hello {x}')
hello 0
>>> x = 1
>>> print(f'hello {x}')
hello 1
>>> x = 2
>>> print(f'hello {x}')
hello 2
```

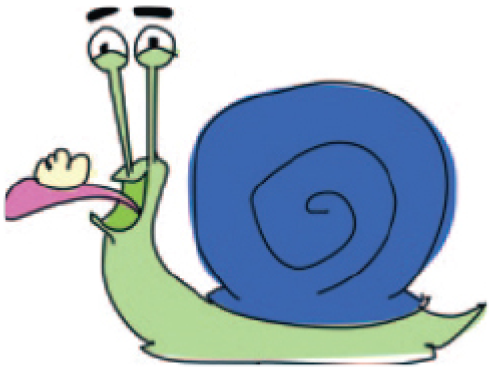
```
>>> x = 3
>>> print(f'hello {x}')
hello 3
>>> x = 4
>>> print(f'hello {x}')
hello 4
```

Using the `for` loop saved us from writing eight extra lines of code! It's best practice to avoid doing things more than once, so the `for` loop is a popular statement among programmers.

You don't need to stick to using the `range` function when making `for` loops. You could also use a list you've already created, such as the shopping list from Chapter 3 , as follows:

```
>>> wizard_list = ['spider legs', 'toe of frog', 'snail tongue',
                   'bat wing', 'slug butter', 'bear burp']
>>> for ingredient in wizard_list:
        print(ingredient)
spider legs
toe of frog
snail tongue
bat wing
slug butter
bear burp
```

This code tells Python, "For each item in `wizard_list` , store the value in the `i` variable, and then print the contents of that variable." If we got rid of the `for` loop, we'd need to do something like this:



```
>>> wizard_list = ['spider legs', 'toe of frog', 'snail tongue',
                   'bat wing', 'slug butter', 'bear burp']
>>> print(wizard_list[0])
spider legs
```

```
>>> print(wizard_list[1])
toe of frog
>>> print(wizard_list[2])
snail tongue
>>> print(wizard_list[3])
bat wing
>>> print(wizard_list[4])
slug butter
>>> print(wizard_list[5])
bear burp
```

Once again, `for` has saved us a lot of typing.

Let's create another loop. Type the following code into the Python Shell; it should automatically indent the code for you:

```
>>> hugehairypants = ['huge', 'hairy', 'pants']
>>> for i in hugehairypants:
        print(i)
        print(i)

huge
huge
hairy
hairy
pants
pants
```

In the first line, we create a list containing `'huge'`, `'hairy'`, and `'pants'`. In the next line, we loop through the items in that list, and then assign each item to the `i` variable. We then print the contents of the variable twice in the next two lines. Press ENTER on the next blank line to tell Python to end the block. It then runs the code and prints each element of the list twice.

Remember that if you enter the wrong number of spaces, you'll end up with an error message. If you entered the preceding code with an extra space on the fourth line, Python would display an indentation error:

```
>>> hugehairypants = ['huge', 'hairy', 'pants']
>>> for i in hugehairypants:
        print(i)
         print(i)

IndentationError: unexpected indent
```

As you learned in Chapter 5 , Python expects the number of spaces in a block to be consistent. It doesn't matter how many spaces you insert, as long as you use the same number for each new line (this also makes the code easier for humans to read).

Here's a more complicated example of a `for` loop with two blocks of code:

```
>>> hugehairypants = ['huge', 'hairy', 'pants']
>>> for i in hugehairypants:
        print(i)
        for j in hugehairypants:
            print(j)
```

Where are the blocks in this code? The first block is the first `for` loop:

```
hugehairypants = ['huge', 'hairy', 'pants']
for i in hugehairypants:
    print(i)                 #
    for j in hugehairypants: # These lines are the FIRST block.
        print(j)             #
```

The next block is the single `print` line in the second `for` loop:

```
hugehairypants = ['huge', 'hairy', 'pants']
for i in hugehairypants:
    print(i)
    for j in hugehairypants:
      print(j)              # This line is also the SECOND block.
```

Can you figure out what this code is going to do?

After a list called `hugehairypants` is created, we can tell from the next two lines that Python is going to loop through the items in the list and print out each one. However, at `for j in hugehairypants` , it will loop over the list again, this time assigning the value to the variable `j` , and then print each item again. These last two lines of code are still part of the first `for` loop, which means they will be executed for each item as the `for` loop goes through the list.

When this code runs, we should see `huge` followed by `huge, hairy, pants` , and then `hairy` followed by `huge, hairy, pants` , and so on.

Enter the code into the Python Shell and see for yourself:

```
   >>> hugehairypants = ['huge', 'hairy', 'pants']
   >>> for i in hugehairypants:
           print(i)
           for j in hugehairypants:
               print(j)

→ huge
   huge
   hairy
   pants
→ hairy
   huge
   hairy
   pants
```

```
→ pants
  huge
  hairy
  pants
```

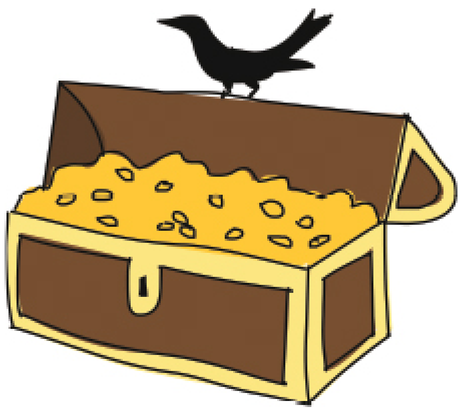
Python enters the first loop and prints an item from the list. Next, it enters the second loop and prints all the items in the list. After that, it prints the second item in the list with `print(i)` , and then prints the complete list again with the `print(j)` command in the inner loop. Lastly, it prints the third item in the list with `print(i)` again, and then prints the complete list one more time with the inner loop. In the output, the lines marked → are printed by `print(i)` , and the unmarked lines are printed by the `print(j)` statement.

Let's try something more practical than printing silly words. Remember the calculation we came up with in Chapter 2 to figure out how many gold coins you'd have at the end of the year if you used your grandfather's duplication machine? It looked like this:

```
>>> found_coins + magic_coins * 365 - stolen_coins * 52
```

This represents 20 found coins plus 10 magic coins multiplied by 365 days in the year, minus the 3 coins a week stolen by the raven.

Let's check how your pile of gold coins will increase each week. We can do this with a `for` loop, but first we need to change the value of our `magic_coins` variable so it represents the total number of coins per week. We get 10 magic coins per day, and there are 7 days in a week, so `magic_coins` should be 70:



```
>>> found_coins = 20
>>> magic_coins = 70
>>> stolen_coins = 3
```

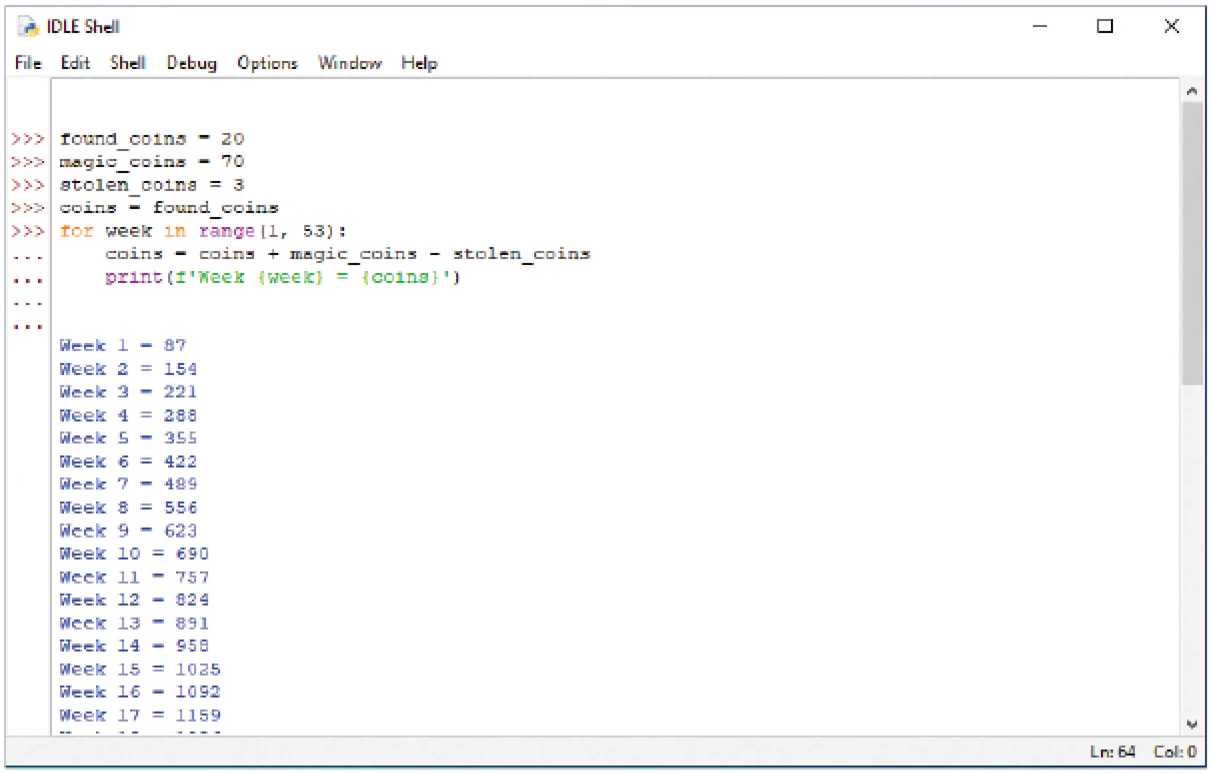
We can see our treasure increase each week by creating another variable, called `coins`, and using a `for` loop:

```
   >>> found_coins = 20
   >>> magic_coins = 70
   >>> stolen_coins = 3
❶ >>> coins = found_coins
   >>> for week in range(1, 53):
           coins = coins + magic_coins - stolen_coins
           print(f'Week {week} = {coins}')
```

The `coins` variable is loaded with the value of the `found_coins` variable ❶; this is our starting number. The next line sets up the `for` loop, which will run the commands in the block. Each time it loops, the `week` variable is loaded with the next number in the range of 1 through 52.

The line containing `coins = coins + magic_coins - stolen_coins` is a bit more complicated. Each week we want to add the number of coins we've magically created and subtract the number of coins the raven has stolen. Think of the `coins` variable as a treasure chest. Every week, the new coins are piled into the chest. So this line is telling Python, "Replace the contents of `coins` with the number of my current coins, plus what I've created this week." The equal sign (`=`) is a bossy piece of code that says, "Work out some stuff on the right first, and then save it for later, using the name on the left."

The `print` statement prints the week number and the total number of coins (so far) to the screen. (Consider rereading "Embedding Values in Strings" on page 29.) If you run this program, you'll see something like Figure 6-1.



```
>>> found_coins = 20
>>> magic_coins = 70
>>> stolen_coins = 3
>>> coins = found_coins
>>> for week in range(1, 53):
...     coins = coins + magic_coins - stolen_coins
...     print(f'Week {week} = {coins}')
...
...

Week 1 = 87
Week 2 = 154
Week 3 = 221
Week 4 = 288
Week 5 = 355
Week 6 = 422
Week 7 = 489
Week 8 = 556
Week 9 = 623
Week 10 = 690
Week 11 = 757
Week 12 = 824
Week 13 = 891
Week 14 = 958
Week 15 = 1025
Week 16 = 1092
Week 17 = 1159
```

Figure 6-1: Running the loop

WHILE WE'RE TALKING ABOUT LOOPING . . .

A `for` loop isn't the only kind of loop you can make in Python. There's also the `while` loop. While a `for` loop has a specific length, a `while` loop doesn't. It's used if you don't know when the loop needs to stop ahead of time.

Imagine a staircase with 20 steps. The staircase is indoors, and you know you can easily climb 20 steps. A `for` loop is like that:

```
>>> for step in range(0, 20):
        print(step)
```

Now imagine a staircase going up a mountainside. The mountain is really tall, and you might run out of energy before you reach the top. Or the weather might turn bad, forcing you to stop. This is what a `while` loop is like:

```
step = 0
while step < 10000:
    print(step)
    if tired == True:
        break
    elif badweather == True:
        break
    else:
        step = step + 1
```

If you try to run this code, you'll get an error, because we haven't created the `tired` and `badweather` variables. Although there isn't enough code here to make a working program, it demonstrates a simple `while` loop.

We start by creating the `step` variable with `step = 0` . Next, we create a `while` loop that checks whether the value of `step` is less than 10,000 (`step < 10000`), which is the number of steps from the bottom of the mountain to the top. As long as `step` is less than 10,000, Python will execute the rest of the code.

With `print(step)` , we print the value of `step` and then check whether the value of the variable `tired` is `True` with the `if tired == True` condition. If it is, we use `break` to exit the loop. The `break` keyword is a way of jumping out of (or stopping) a loop immediately, and it works with both `while` and `for` loops.

In this example, `break` causes Python to jump out of the block and move to any commands that appeared after the `step = step + 1` line.

The line `elif badweather == True` checks to see if `badweather` is set to `True` ; if so, `break` exits the loop. If neither `tired` nor `badweather` is `True` (seen at `else`), we add 1 to the `step` variable with `step = step + 1` , and the loop continues.

The steps of a `while` loop are as follows:

1. Check the condition.
2. Execute the code in the block.
3. Repeat.

More commonly, a `while` loop might be created with a few conditions, rather than just one, like this:

```
>>> x = 45
>>> y = 80
>>> while x < 50 and y < 100:
        x = x + 1
        y = y + 1
        print(x, y)
```

Here, we create an `x` variable with the value 45 and a `y` variable with the value 80. The loop checks for two conditions: whether `x` is less than 50 and whether `y` is less than 100. While both conditions are true, the lines that follow are executed, adding 1 to both variables and then printing them. The output of this code is as follows:

```
46 81
47 82
48 83
49 84
50 85
```

Can you figure out how this works?

We start counting at 45 for `x` and at 80 for `y` , and then *increment* (add 1 to each variable) every time the code in the loop is run. The loop will run as long as `x` is less than 50 and `y` is less than 100. After looping five times, the value in `x` reaches 50. Now the first condition (`x < 50`) is no longer true, so Python stops looping.

We can also use a `while` loop to create a *semi-eternal loop* that could go on forever, but continues until something happens in the code to break out of it.

Here's an example:

```
while True:
    lots of code here
    lots of code here
    lots of code here
    if some_value == True:
        break
```

The condition for the `while` loop is just `True` , which is always true, so the code in the block will always run (thus, the loop is eternal). Python will break out of the loop only if the variable `some_value` is true.

WHAT YOU LEARNED

In this chapter, we used two types of loops to perform repetitive tasks: `for` loops and `while` loops. These are similar but can be used in different ways. We told Python what we wanted repeated by writing the tasks inside blocks of code, which we put inside loops. We also used the `break` keyword to stop looping.

PROGRAMMING PUZZLES

Here are some examples of loops to try out. The solutions can be found at *http://python-for-kids.com* .

#1: THE HELLO LOOP

What do you think the following code will do? Guess what will happen, and then run the code in Python to see if you're right.

```
>>> for x in range(0, 20):
        print(f'hello {x}')
        if x < 9:
           break
```

#2: EVEN NUMBERS

Create a loop that prints even numbers until it reaches your age (if your age is an odd number, create a loop that prints out odd numbers until it reaches your age). For example, it might print out something like this:

```
2
4
6
8
10
12
14
```

#3: MY FIVE FAVORITE INGREDIENTS

Create a list containing five different sandwich ingredients, such as the following:

```
>>> ingredients = ['snails', 'leeches', 'gorilla belly-button lint',
                   'caterpillar eyebrows', 'centipede toes']
```

Now create a loop that prints out the list (including the numbers):

```
1 snails
2 leeches
3 gorilla belly-button lint
4 caterpillar eyebrows
5 centipede toes
```

#4: YOUR WEIGHT ON THE MOON

If you were standing on the moon right now, your weight would be 16.5 percent of what it is on Earth. You can calculate that by multiplying your Earth weight by 0.165.

If you gained two pounds every year for the next 15 years, what would your weight be when you visited the moon each year and at the end of the 15 years? Write a program using a `for` loop that prints your moon weight for each year.

7
RECYCLING YOUR CODE WITH FUNCTIONS AND MODULES

Think about how much stuff you throw away each day: water bottles, soda cans, potato chip bags, plastic sandwich wrappers, bags that held carrot sticks or apple slices, shopping bags, newspapers, magazines, and so on. Now imagine all of that trash just got dumped in a pile at the end of your driveway, without separating out the paper, the plastic, and the tin cans.

You probably recycle as much as possible, which is good, because no one likes to climb over a pile of trash on the way to school. Rather than sitting in an enormous, gross pile, recycled glass bottles are melted down and turned into new jars and bottles; paper is pulped into recycled paper; and plastic is turned into heavier plastic goods. We reuse things we would otherwise throw away.

In the programming world, recycling is just as important. Your program might not disappear under a pile of garbage, but if you never reuse any code, you'll type so much that your fingers might eventually wear down to painful stubs! Recycling also makes your code shorter and easier to read.

As you'll learn in this chapter, Python offers a number of ways to reuse code.

USING FUNCTIONS

Functions are chunks of code that tell Python to do something. They are one way to reuse code—you can use functions in your programs again and again. Python has many functions available to use; these are called *built-in* functions, or *builtins* (for more information on builtins, see Appendix B). There are also functions available in modules (more on those below), and you can even write functions yourself.

We started learning about functions in the previous chapter, when we used `range` and `list` to make Python count:

```
>>> list(range(0, 5))
[0, 1, 2, 3, 4]
```

It's not too hard to type a list of consecutive numbers yourself, but the larger the list, the more you need to type. With functions, you can just as easily create a list of a thousand numbers.

Here's an example that uses the `list` and `range` functions to produce a list of numbers:

```
>>> list(range(0, 1000))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16..., 997, 998, 999]
```

When you're writing simple programs, functions are handy. Once you start writing long, complex programs—like games—functions are *essential* (assuming you want to finish writing your program this century).

Let's look at how to write our own functions.

PARTS OF A FUNCTION

A function has three parts: a name, parameters, and a body. Here's an example of a simple function:

```
>>> def testfunc(myname):
        print(f'hello {myname}')
```

The name of this function is `testfunc` . It has a single parameter, `myname` , and its body is the block of code immediately following the line beginning with `def` (short for *define*). A *parameter* is a variable that exists only while a function is being used.

You can run the function by calling its name, using parentheses around the parameter value:

```
>>> testfunc('Mary')
hello Mary
```

Functions can take any number of parameters:

```
>>> def testfunc(fname, lname):
        print(f'Hello {fname} {lname}')
```

When using multiple parameters, make sure you separate the values with a comma:

```
>>> testfunc('Mary', 'Smith')
Hello Mary Smith
```

We can also create variables first and then call the function with them:

```
>>> firstname = 'Joe'
>>> lastname = 'Robertson'
>>> testfunc(firstname, lastname)
Hello Joe Robertson
```

A function can return a value with a `return` statement. For example, you could write a function to calculate how much money you're saving:

```
>>> def savings(pocket_money, paper_route, spending):
        return pocket_money + paper_route - spending
```

This function takes three parameters. It adds the first two (`pocket_money` and `paper_route`) and subtracts the last (`spending`). The result is returned and can be assigned to a variable (the same way we set other values to variables) or printed:

```
>>> print(savings(10, 10, 5))
15
```

We pass in parameters `10` , `10` , and `5` , and the `savings` function calculates `15` as the result and then returns the value.

VARIABLES AND SCOPE

A variable that's inside the body of a function can't be used again once the function has finished running because it exists only inside the function. In the world of programming, where a variable can be used is called its *scope* . Let's look at a simple function that uses a couple of variables but doesn't have any parameters:

```
>>> def variable_test():
        first_variable = 10
        second_variable = 20
        return first_variable * second_variable
```

In this example, we create the `variable_test` function, which multiplies `first_variable` and `second_variable` and returns the result:

```
>>> print(variable_test())
200
```

If we call this function using `print` , we get 200. However, if we try to print the contents of `first_variable` (or `second_variable` , for that matter) outside of the block of code in the function, we get an error message:

```
>>> print(first_variable)
Traceback (most recent call last):
  File "<pyshell#50>", line 1, in <module>
    print(first_variable)
NameError: name 'first_variable' is not defined
```

If a variable is defined outside the function, it has a different scope. For example, let's define a variable before we create our function, and then try using it inside the function:

```
❶ >>> another_variable = 100
  >>> def variable_test2():
          first_variable = 10
          second_variable = 20
      ❷ return first_variable * second_variable * another_variable
```

In this code, even though the variables `first_variable` and `second_variable` can't be used outside the function, the variable `another_variable` (which was created outside the function ❶) can be used inside it ❷ .

Here's the result of calling this function:

```
>>> print(variable_test2())
20000
```

Now, suppose you were building a spaceship out of something economical like recycled tin cans. You can flatten two cans a week to create the curved walls of your spaceship, but you'll need about 500 cans to finish the fuselage. Let's try writing a function to print out the total cans flattened each week over a year.

Our function will calculate how many cans we've flattened each week, with the number of cans as a parameter (that makes it easier to change the number of cans later):

```
>>> def spaceship_building(cans):
        total_cans = 0
        for week in range(1, 53):
            total_cans = total_cans + cans
            print(f'Week {week} = {total_cans} cans')
```

On the first line of the function, we create the `total_cans` variable and set its value to 0. We then create a loop for the weeks in the year and add the number of cans flattened each week. This block of code makes up the content of our function, and the last two lines make up another block of the `for` loop.

Let's try entering that function in the Python Shell and calling it with different values for the number of cans, starting with 2:

```
>>> spaceship_building(2)
Week 1 = 2 cans
Week 2 = 4 cans
Week 3 = 6 cans
Week 4 = 8 cans
Week 5 = 10 cans
Week 6 = 12 cans
Week 7 = 14 cans
Week 8 = 16 cans
...
Week 50 = 100 cans
Week 51 = 102 cans
Week 52 = 104 cans
```

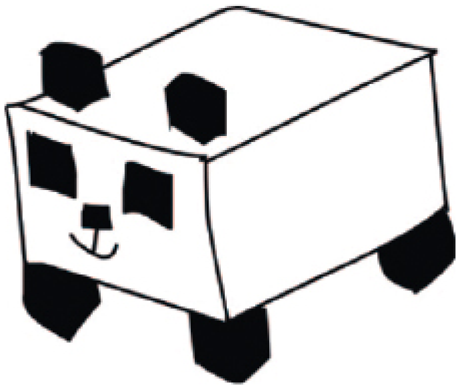
```
>>> spaceship_building(10)
Week 1 = 10 cans
Week 2 = 20 cans
Week 3 = 30 cans
Week 4 = 40 cans
Week 5 = 50 cans
...
Week 48 = 480 cans
Week 49 = 490 cans
Week 50 = 500 cans
Week 51 = 510 cans
Week 52 = 520 cans
```

This function can be reused with different values for the number of cans per week, which is a bit more efficient than retyping the `for` loop every time you want to try it with different numbers. When we run `spaceship_building(10)` , we can see we'll have enough cans to build the spaceship walls at week 50.

Functions can also be grouped together into modules, which makes Python *really* useful, as opposed to just mildly useful.

USING MODULES

Modules are used to group functions, variables, and other things together into larger, more powerful programs. Some modules are built in to Python, and others must be downloaded separately. There are modules to help you write games (such as `tkinter` , which is built in, and `PyGame` , which is not), modules for manipulating images (such as `Pillow` , the Python Imaging Library), and modules for drawing three-dimensional graphics (such as `Panda3D`).



Modules can be used to do all sorts of useful things. For example, if you were designing a simulation game and you wanted the world to change according to the real world, you could calculate the current date and time by using the built-in `time` module:

```
>>> import time
```

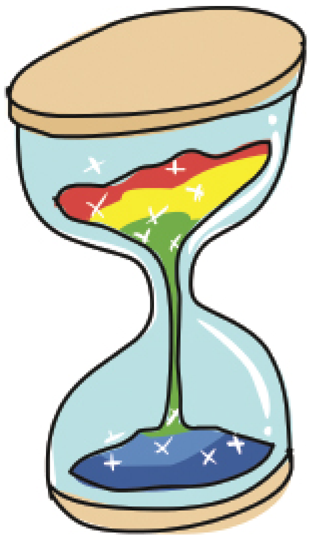
The `import` command tells Python we want to use the `time` module.

We can then call functions available in this module by using the dot symbol. (We used similar functions to work with the `turtle` module in Chapter 4 , such as `t.forward(50)` .) For example, here's how we might call the `asctime` function in the `time` module:

```
>>> print(time.asctime())
Tue Aug 12 07:05:32 2025
```

The `asctime` function is a part of the `time` module that returns the current date and time as a string.

Now suppose you want to ask someone to enter a value, like their date of birth or age. You can do this using a `print` statement to display a message, and the `sys` (short for *system*) module, which contains utilities for interacting with the Python system itself. First, we import the `sys` module:



```
>>> import sys
```

Inside the `sys` module is a special object called `stdin` (for *standard input*), which provides a useful function called `readline` . The `readline` function is used to read a line of text typed on the keyboard until you press ENTER. (We'll look at how objects work in Chapter 8 .) To test `readline` , enter the following code in the Python Shell:

```
>>> import sys
>>> print(sys.stdin.readline())
```

If you then type some words and press ENTER, those words will be printed out in the Python Shell.

Think back to the code we wrote in Chapter 5 , using an `if` statement:

```
>>> if age >= 10 and age <= 13:
        print('What is 13 + 49 + 84 + 155 + 97? A headache!')
    else:
        print('Huh?')
```

Rather than creating the `age` variable and giving it a specific value before the `if` statement, we can now ask someone to enter the value instead. But first, let's turn the code into a function:

```
>>> def silly_age_joke(age):
        if age >= 10 and age <= 13:
            print('What is 13 + 49 + 84 + 155 + 97? A headache!')
        else:
            print('Huh?')
```

Now you can call the function by entering its name and telling it what number to use by entering the number in parentheses. Does it work?

```
>>> silly_age_joke(9)
Huh?
>>> silly_age_joke(10)
What is 13 + 49 + 84 + 155 + 97? A headache!
```

It works! Now let's make the function ask for a person's age. (You can add to or change a function as many times as you want.)

```
>>> def silly_age_joke():
        print('How old are you?')
     ❶ age = int(sys.stdin.readline())
     ❷ if age >= 10 and age <= 13:
            print('What is 13 + 49 + 84 + 155 + 97? A headache!')
        else:
            print('Huh?')
```

Did you recognize the function `int` ❶ , which converts a string to a number? We included that function because `sys.stdin. readline()` returns whatever someone enters as a string, but we want a number so we can compare it with the numbers 10 and 13 ❷ . To try this yourself, call the function without any parameters, and then type a number when `How old are you?` appears:

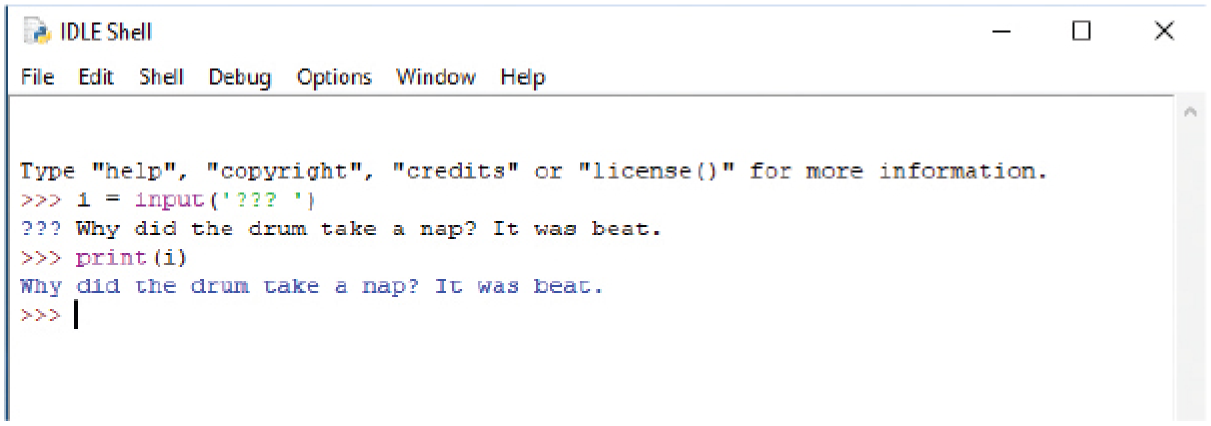
```
>>> silly_age_joke()
How old are you?
10
What is 13 + 49 + 84 + 155 + 97? A headache!
>>> silly_age_joke()
How old are you?
15
Huh?
```

The first time we call the function, it displays "How old are you?"; then we enter `10` , and it prints the joke. The second time, we enter `15` and it prints “Huh?”

THE INPUT FUNCTION

The `sys.stdin.readline` function is not the only way to read input from the keyboard. A much simpler option is the built-in `input` function. The `input` function takes an optional `prompt` parameter (a string with the message you want to display), and then returns whatever is typed until the ENTER key is hit. Figure 7-1 shows what happens when you run this code:

```
i = input('??? ')
print(i)
```



```
Type "help", "copyright", "credits" or "license()" for more information.
>>> i = input('??? ')
??? Why did the drum take a nap? It was beat.
>>> print(i)
Why did the drum take a nap? It was beat.
>>> 
```

Figure 7-1: Using the `input` function

Let’s rewrite the `silly_age_joke` function to use `input` instead:

```
>>> def silly_age_joke():
        age = int(input('How old are you?\n'))
        if age >= 10 and age <= 13:
            print('What is 13 + 49 + 84 + 155 + 97? A headache!')
        else:
            print('Huh?')
```

Apart from having slightly fewer lines, another difference between the earlier code and this version is that we add a *newline* character (\ `n`) at the end of our string (‘`How old are you?` \ `n`’). A newline simply moves the cursor from one line of the display down to the next one. This happens automatically with `print` , but not with the `input` function.

Regardless, this code works exactly the same as before.

WHAT YOU LEARNED

In this chapter, you learned to make reusable chunks of code in Python with functions and use functions provided by modules. You saw how the scope of variables controls whether they can be seen inside or outside of functions. You also learned how to create functions by using the `def` keyword and how to import modules to use their contents.

PROGRAMMING PUZZLES

Try the following examples to experiment with creating your own functions. The solutions can be found at *http://python-for-kids.com* .

#1: BASIC MOON WEIGHT FUNCTION

In one of Chapter 6 's programming puzzles, we created a `for` loop to determine your weight on the moon over a period of 15 years. That `for` loop could easily be turned into a function. Try creating a function that takes a starting weight and increases its amount each year. You might call the new function using code like this:

```
>>> moon_weight(30, 0.25)
```

#2: MOON WEIGHT FUNCTION AND YEARS

Take the function you've just created and change it to work out the weight over different periods, such as 5 years or 20 years. Be sure to change the function so it takes three arguments: the initial weight, the weight gained each year, and the number of years:

```
>>> moon_weight(90, 0.25, 5)
```

#3: MOON WEIGHT PROGRAM

Instead of using a simple function and passing in the values as parameters, you can use `sys.stdin.readline()` or `input()` to make a mini-program that prompts for the values. In this case, you call the function without any parameters at all:

```
>>> moon_weight()
```

The function will display a message asking for the starting weight, a second message asking for the amount the weight will increase each year, and a final message asking for the number of years. You would see something like the following:

```
Please enter your current Earth weight
45
Please enter the amount your weight might increase each year
0.4
Please enter the number of years
12
```

Remember to import the `sys` module before creating your function, if you are using `sys.stdin.readline()` :

```
>>> import sys
```

#4: MARS WEIGHT PROGRAM

Let's change our moon weight program to calculate weights on Mars—only this time, for your entire family. The function should ask for each family member's weight, calculate how much they would weigh on Mars (by multiplying the number by 0.3782), and then add up and display the total weight at the end. You can write this code in many ways; what's important is that it displays the total weight at the end.

8
HOW TO USE CLASSES AND OBJECTS

Why is a giraffe like a sidewalk? Because a giraffe and a sidewalk are both *things* , which are known in the English language as nouns and in Python as *objects* . In programming, objects are a way to organize code and break things down to more easily work with complex ideas. (We used an object in Chapter 4 when we worked with the `turtle` module's `Turtle` object.)

To fully understand how objects work in Python, we need to think about types of objects. Let's start with giraffes and sidewalks.

A giraffe is a type of mammal, which is a type of animal. A giraffe is also an animate object—it's alive.

There's not much to say about a sidewalk other than it's not a living thing. Let's call it an inanimate object (in other words, it's not alive). The terms *mammal* , *animal* , *animate* , and *inanimate* are all ways of classifying things.

BREAKING THINGS INTO CLASSES

In Python, objects are defined by *classes* , which classify objects into groups. For example, the tree diagram in Figure 8-1 shows the classes that giraffes and sidewalks fit into based on our preceding definitions.

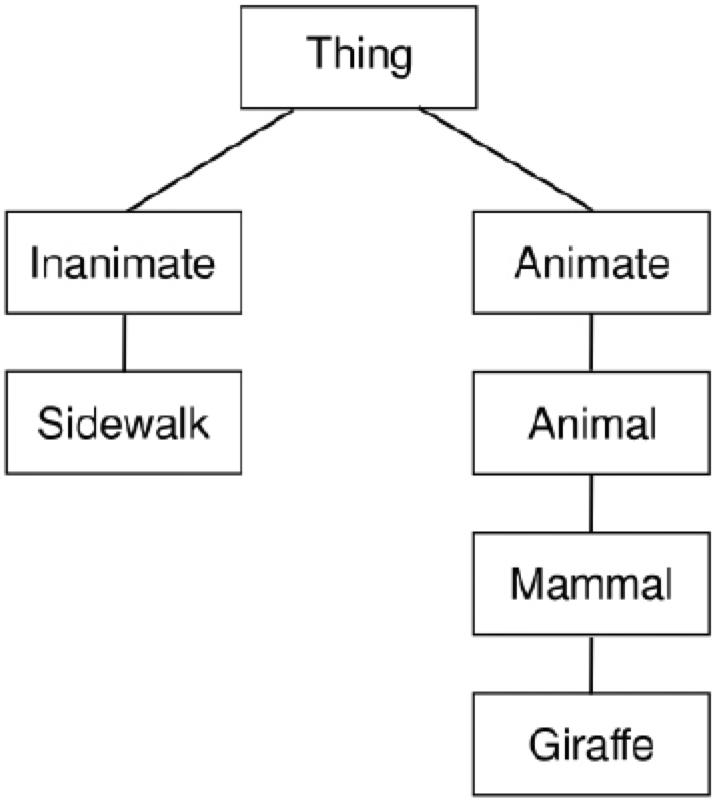


Figure 8-1: Tree diagram of some classes

The main class is `Thing` . Below the `Thing` class, we have `Inanimate` and `Animate` . These classes are further broken down into `Sidewalk` for `Inanimate` , and `Animal` , `Mammal` , and `Giraffe` for `Animate` .

We can use classes to organize bits of code. For example, consider the `turtle` module. All the things Python's turtle module can do—such as moving forward, moving backward, turning left, and turning right—are functions in the `Turtle` class. An object is a member of a class, and we can create any number of objects for a class, which we'll get to shortly.

Now let's create the same set of classes shown in our tree diagram, starting from the top. We define classes using the `class` keyword followed by a name. `Thing` is the broadest class, so we'll create it first:

```
>>> class Thing:
        pass
```

We name the class `Thing` and use the `pass` statement to let Python know we're not going to give any more information. The `pass` keyword is used when we want to provide a class or function but don't want to fill in the details at the moment.

Next, we'll add the other classes and build some relationships between them.

CHILDREN AND PARENTS

If a class is a part of another class, it's considered a *child* of that class, and the other class is its *parent* . Classes can be both *children of* and *parents to* other classes. In our tree diagram, the class above another class is its parent and the class below it is its child. For example, `Inanimate` and `Animate` are both children of the class `Thing` , meaning that `Thing` is their parent.

To tell Python that a class is a child of another class, we add the name of the parent class in parentheses after the name of our new class, like this:

```
>>> class Inanimate(Thing):
        pass

>>> class Animate(Thing):
        pass
```

Here, we create a class called `Inanimate` and tell Python that its parent class is `Thing` . Next, we create a class called `Animate` and tell Python that its parent class is also `Thing` .

Let's create the `Sidewalk` class with the parent class `Inanimate` :

```
>>> class Sidewalk(Inanimate):
        pass
```

We can organize the `Animal` , `Mammal` , and `Giraffe` classes using their parent classes as well:

```
>>> class Animal(Animate):
        pass

>>> class Mammal(Animal):
        pass

>>> class Giraffe(Mammal):
        pass
```

ADDING OBJECTS TO CLASSES

We now have a bunch of classes, but what about putting some more information into those classes? Say we have a giraffe named Reginald. We know he belongs in the `Giraffe` class, but what do we use—in programming terms—to describe the single giraffe called Reginald?

We call Reginald an *object* (also known as an *instance*) of the `Giraffe` class. To "introduce" Reginald to Python, we use this little snippet of code:

```
>>> reginald = Giraffe()
```

This code tells Python to create an object of the `Giraffe` class and assign it to the `reginald` variable. Like when we call a function, the class name is followed by parentheses. Later in this chapter, we'll see how to create objects and use parameters in the parentheses.

But what does the `reginald` object do? Well, nothing at the moment. To make our objects useful, when we create our classes, we also need to define functions that can be used with the objects in that class. Rather than just using the `pass` keyword immediately after defining the class, we can add function definitions.

DEFINING FUNCTIONS OF CLASSES

Chapter 7 introduced functions as a way to reuse code. When we define a function that's associated with a class, we do so in the same way that we define any other function, except we indent it beneath the class definition. For example, here's a normal function that isn't associated with a class:

```
>>> def this_is_a_normal_function():
        print('I am a normal function')
```

And here are a couple of functions that are defined for a class:

```
>>> class ThisIsMySillyClass:
        def this_is_a_class_function():
            print('I am a class function')
        def this_is_also_a_class_function():
            print('I am also a class function. See?')
```

ADDING CLASS CHARACTERISTICS

Consider the child classes of the `Animate` class we defined in section Children and Parents. We can add characteristics to each class that describe what it is and what it can do. A *characteristic* is a trait all members of the class (and its children) share.

For example, what do all animals have in common? To start with, they breathe, move, and eat. What about mammals? Mammals feed their young with milk, and they also breathe, move, and eat. We know giraffes eat leaves from high up in trees. And like all mammals, they feed their young with milk, breathe, move, and eat. When we add these characteristics to our tree diagram, we get something like Figure 8-2 .

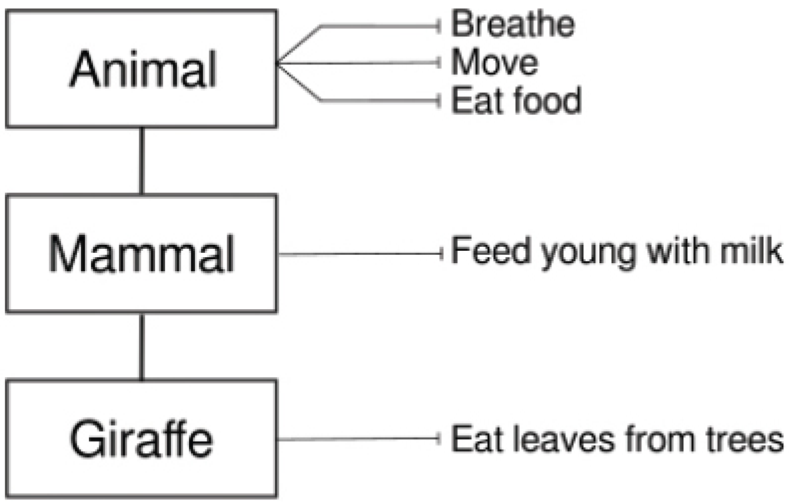


Figure 8-2: Classes with characteristics

These characteristics can be thought of as actions, or functions—things that an object of that class can do.

To add a function to a class, we use the `def` keyword. So the `Animal` class will look like this:

```
>>> class Animal(Animate):
        def breathe(self):
            pass
        def move(self):
            pass
        def eat_food(self):
            pass
```

In the first line of this listing, we define the class as we did before, but instead of using `pass` on the next line, we define a function called `breathe` and give it the `self` parameter. The `self` parameter is a way for one function in the class to call another function in the class (and in the parent class). We'll see this parameter in use later.

On the next line, `pass` tells Python we're not going to provide any more information about the `breathe` function because it's going to do nothing for now. Then we add the `move` and `eat_food` functions and use the `pass` keyword for both. We'll recreate our classes shortly and put some proper code in the functions. This is a common way to develop programs.

NOTE

Often, programmers will create classes with functions that do nothing as a way to figure out what the class should do before getting into the details of the individual functions.

We can also add functions to the `Mammal` and `Giraffe` classes. Each class will be able to use the characteristics (or functions) of its parent, meaning you don't need to make one really complicated class. Instead, you can put your functions in the highest parent class where the characteristic applies. (This makes your classes simpler and easier to understand.)

```
>>> class Mammal(Animal):
        def feed_young_with_milk(self):
            pass
```

```
>>> class Giraffe(Mammal):
        def eat_leaves_from_trees(self):
            pass
```

In the above code, the `Mammal` class provides a function `feed _young_with_milk` . The `Giraffe` class is a child class (or *subclass*) of `Mammal` and provides another function: `eat_leaves_from_trees` .

WHY USE CLASSES AND OBJECTS?

We've now added functions to our classes, but why use classes and objects at all when you could just write normal functions called `breathe` , `move` , `eat_food` , and so on?

To answer that question, we'll use our giraffe called Reginald, which we created earlier as an object of the `Giraffe` class, like this:

```
>>> reginald = Giraffe()
```

Because `reginald` is an object, we can call (or run) functions provided by the `Giraffe` class and its parent classes. We call functions on an object by using the dot (.) operator and the name of the function. To tell Reginald to move or eat, we can call the functions like this:

```
>>> reginald = Giraffe()
>>> reginald.move()
>>> reginald.eat_leaves_from_trees()
```

Suppose Reginald has a giraffe friend named Harriet. Let's create another `Giraffe` object called `harriet` :

```
>>> harriet = Giraffe()
```

Because we're using objects and classes, we can tell Python which giraffe we're talking about when we want to run the `move` function. For example, if we want to make Harriet move but leave Reginald in place, we could call the `move` function by using our `harriet` object, like this:

```
>>> harriet.move()
```

In this case, only Harriet would be moving.

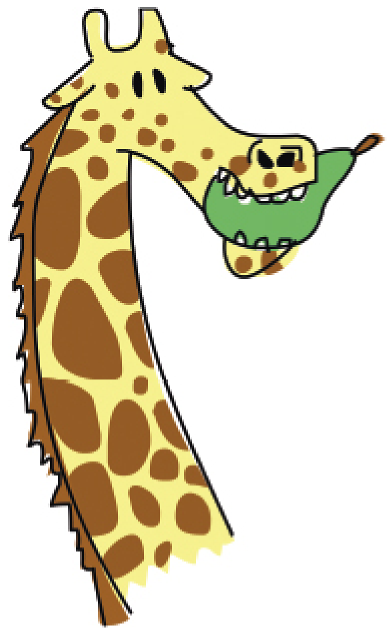
Let's change our classes a little to make this more obvious. Add a `print` statement to each function instead of using `pass` :

```
>>> class Animal(Animate):
        def breathe(self):
            print('breathing')
        def move(self):
            print('moving')
        def eat_food(self):
            print('eating food')
>>> class Mammal(Animal):
        def feed_young_with_milk(self):
            print('feeding young')

>>> class Giraffe(Mammal):
        def eat_leaves_from_trees(self):
            print('eating leaves')
```

Now when we create our `reginald` and `harriet` objects and call functions on them, we can see something actually happen:

```
>>> reginald = Giraffe()
>>> harriet = Giraffe()
>>> reginald.move()
moving
>>> harriet.eat_leaves_from_trees()
eating leaves
```

On the first two lines, we create the variables `reginald` and `harriet` , which are objects of the `Giraffe` class. Next, we call the `move` function on `reginald` , and Python prints `moving` on the following line. In the same way, we call the `eat_leaves_from_trees` function on `harriet` , and Python prints `eating leaves` . If these were real giraffes, rather than objects in a computer, one giraffe would be walking and the other would be eating.



NOTE

Functions defined for classes are actually called `methods` *. The terms are almost interchangeable except that methods can only be called on objects of a class. Another way of saying this is that a method is associated with a class, but a function is not. Given they are almost the same, we'll use the term* `function` *in this book.*

OBJECTS AND CLASSES IN PICTURES

Let's try taking a more graphical approach to objects and classes and return to the `turtle` module we toyed with in Chapter 4 . When we use `turtle.Turtle()` , Python creates an object of the `Turtle` class that is provided by the `turtle` module (similar to our `reginald` and `harriet` objects). We can create two `Turtle` objects (named Avery and Kate) just as we created two giraffes:

```
>>> import turtle
>>> avery = turtle.Turtle()
>>> kate = turtle.Turtle()
```

Each turtle object (`avery` and `kate`) is a member of the `Turtle` class.

Now here's where objects start to become powerful. Having created our `Turtle` objects, we can call functions on each, and they will draw independently. Try this code:

```
>>> avery.forward(50)
>>> avery.right(90)
>>> avery.forward(20)
```

With this series of instructions, we tell Avery to move forward 50 pixels, turn right 90 degrees, and move forward 20 pixels so she finishes facing downward. Remember that turtles always start off facing to the right.

Now it's time to move Kate.

```
>>> kate.left(90)
>>> kate.forward(100)
```

We tell Kate to turn left 90 degrees and then move forward 100 pixels so she ends facing up. So far, we have a line with arrowheads moving in two different directions, with the head of each arrow representing a different turtle object: Avery is pointing down, and Kate is facing up (see Figure 8-3).

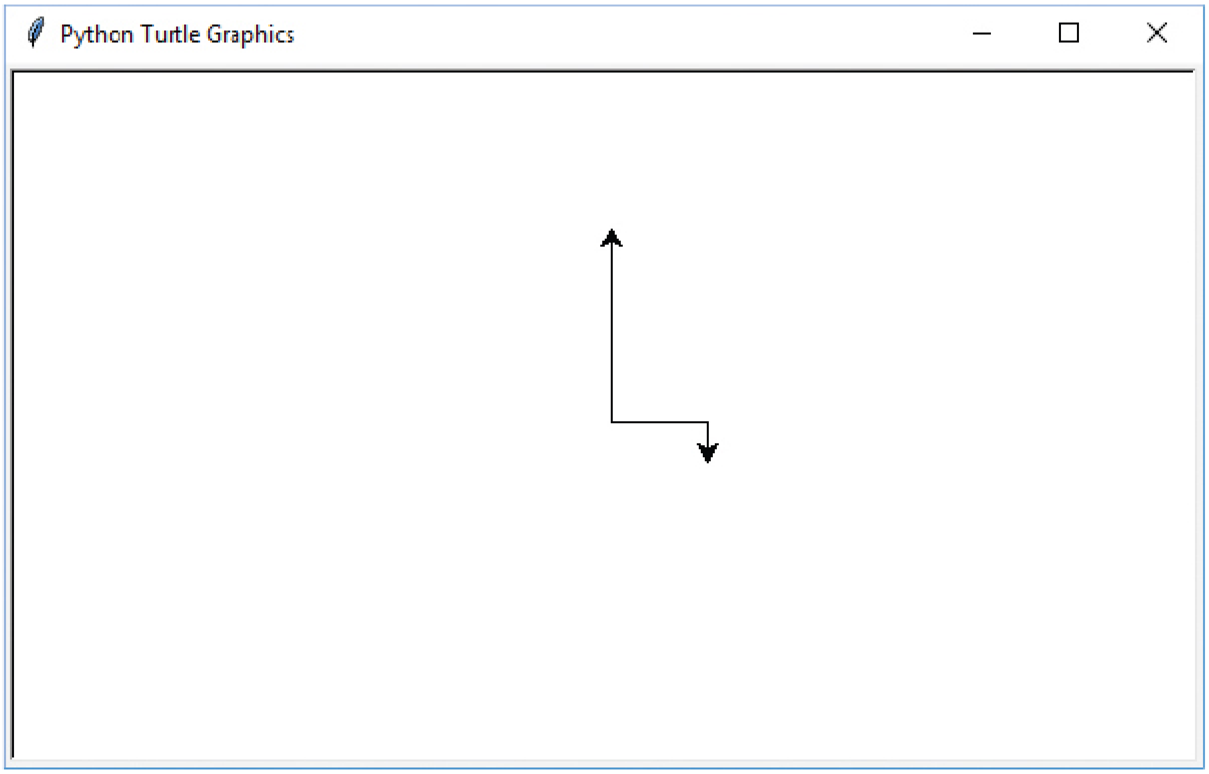
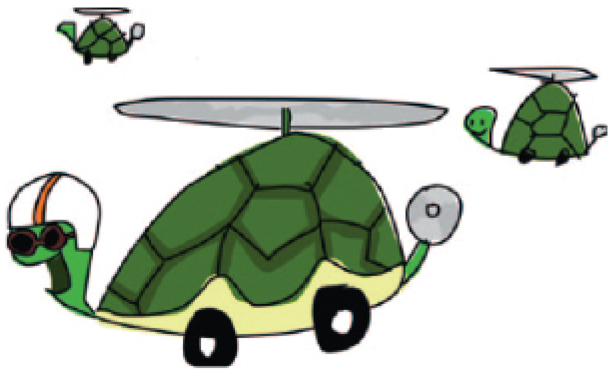


Figure 8-3: Kate and Avery

Now let's add another turtle, Jacob, and move him without bugging Kate or Avery:

```
>>> jacob = turtle.Turtle()
>>> jacob.left(180)
>>> jacob.forward(80)
```



First, we create a new `Turtle` object called `jacob` ; then we turn him left 180 degrees and move him forward 80 pixels. Our drawing with three turtles should look like Figure 8-4 .

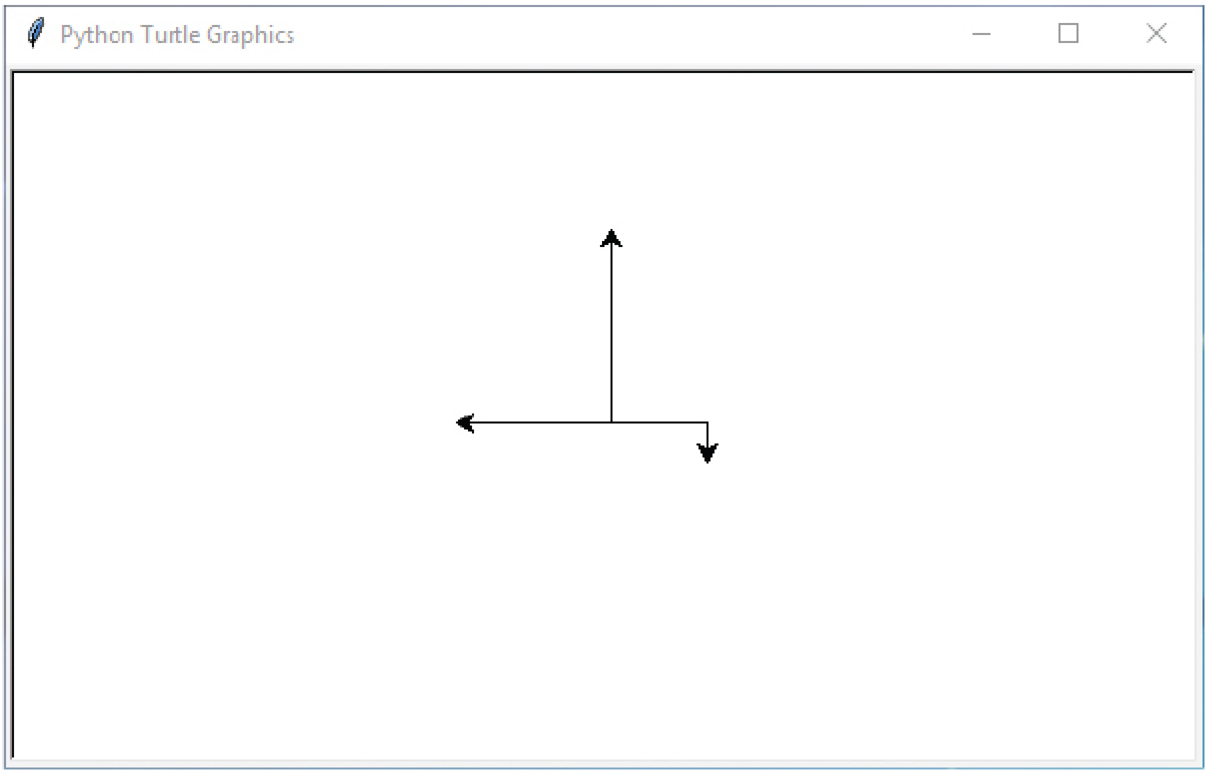


Figure 8-4: Kate and Avery and Jacob

Every time we call `turtle.Turtle()` to create a turtle, we add a new, independent object. Each object is still an instance of the `Turtle` class, and we can use the same functions on each object. But because we're using objects, we can move each turtle independently. Like our independent `Giraffe` objects (Reginald and Harriet), Avery, Kate, and Jacob are independent `Turtle` objects. If we create a new object with the same variable name as an object we've already created, the old object won't necessarily vanish.

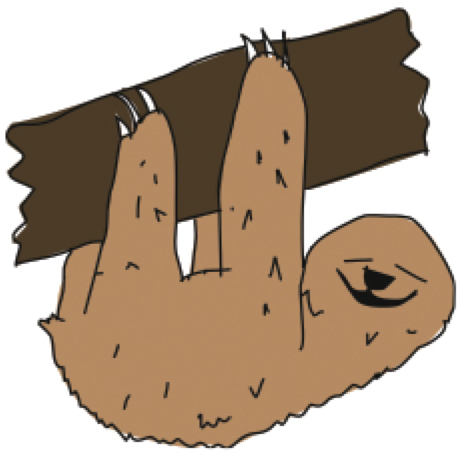
OTHER USEFUL FEATURES OF OBJECTS AND CLASSES

Classes and objects make it easy to group functions. They're also really useful when we want to think about a program in smaller chunks.

For example, consider a huge software application like a word processor or a 3D computer game. It's nearly impossible for most people to fully understand large programs like these because there's so much code. But break these monster programs into smaller pieces, and each piece starts to make sense—as long as you know its programming language, of course!

When writing a large program, breaking it up also allows you to divide the work among other programmers. The most complicated programs (like your web browser) were written by many people, or teams of people, working on different parts at the same time around the world.

Imagine we want to expand some of the classes we've created in this chapter (`Animal` , `Mammal` , and `Giraffe`), but we have too much work to do, and our friends offer to help. We could divide the work of writing the code so that one person works on the `Animal` class, another on the `Mammal` class, and still another on the `Giraffe` class.



INHERITED FUNCTIONS

You may realize that whoever ends up working on the `Giraffe` class is lucky, because any functions created by the people working on the `Animal` and `Mammal` classes can also be used by the `Giraffe` class. The `Giraffe` class *inherits* functions from the `Mammal` class, which, in turn, inherits functions from the `Animal` class. In other words, when we create a `Giraffe` object, we can use functions defined in the `Giraffe` class, as well as functions defined in the `Mammal` and `Animal` classes. And, by the same token, if we create a `Mammal` object, we can use functions defined in the `Mammal` class as well as its parent class, `Animal` .

Look at the relationships between the `Animal` , `Mammal` , and `Giraffe` classes again. The `Animal` class is the parent of the `Mammal` class, and `Mammal` is the parent of the `Giraffe` class (Figure 8-5).

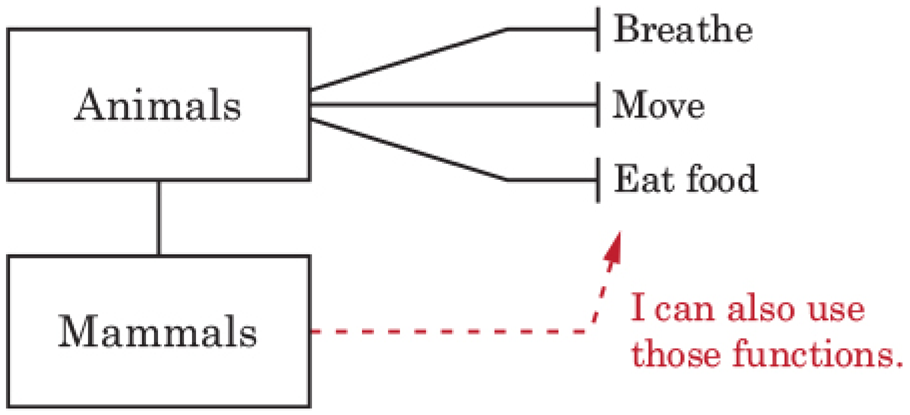


Figure 8-5: Classes and inherited functions

Even though `reginald` is an object of the `Giraffe` class, we can still call the `move` function we defined in the `Animal` class because functions defined in any parent class are available to its child classes:

```
>>> reginald = Giraffe()
>>> reginald.move()
moving
```

In fact, all the functions we defined in both the `Animal` and `Mammal` classes can be called from our `reginald` object because the functions are inherited:

```
>>> reginald = Giraffe()
>>> reginald.breathe()
breathing
>>> reginald.eat_food()
eating food
>>> reginald.feed_young_with_milk()
feeding young
```

In this code, we create an object of the `Giraffe` class called `reginald` . When we call each function, it prints a message regardless of whether the function is defined in `Giraffe` or in a parent class.

FUNCTIONS CALLING OTHER FUNCTIONS

When we call functions on an object, we use the object's variable name. For example, we can call the `move` function on Reginald the giraffe like so:

```
>>> reginald.move()
```

To have a function in the `Giraffe` class call the `move` function, we'd use the `self` parameter. The `self` parameter is a way for one function in the class to call another function. For example, suppose we add a function called `find_food` to the `Giraffe` class:

```
>>> class Giraffe(Mammal):
        def find_food(self):
            self.move()
            print('I\'ve found food!')
            self.eat_food()
```

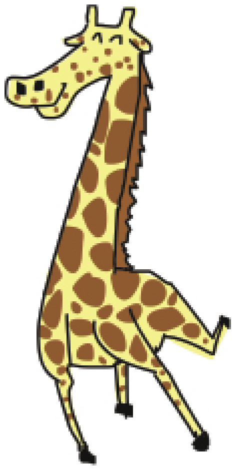
We've created a function that combines two other functions, which is quite common in programming. Often, you'll write a function that does something useful, which you can then use inside another function. (We'll do this in Chapter 11 , where we'll write more complex functions to create a game.)

Let's use `self` to add some functions to the `Giraffe` class:

```
>>> class Giraffe(Mammal):
        def find_food(self):
            self.move()
            print('I\'ve found leaves!')
            self.eat_food()
        def eat_leaves_from_trees(self):
            print('tear leaves from branches')
            self.eat_food()
        def dance_a_jig(self):
```

```
        self.move()
        self.move()
        self.move()
        self.move()
```

We use the eat_food and move functions from the parent Animal class to define eat _leaves_from_trees and dance_a_jig for the Giraffe class, because these are inherited functions. By adding functions that call other functions, when we create objects of these classes, we can call a single function that does more than just one thing. See what happens when we call the dance_a_jig function:



```
>>> reginald = Giraffe()
>>> reginald.dance_a_jig()
moving
moving
moving
moving
```

In this code, our giraffe moves four times (that is, the text moving is printed four times).

If we call the find_food function, we get three lines printed:

```
>>> reginald.find_food()
moving
I've found leaves!
eating food
```

INITIALIZING AN OBJECT

Sometimes when creating an object, we want to set some values (also called *properties*) for later use. When we *initialize* an object, we're getting it ready to be used.

For example, suppose we want to set the number of spots on our giraffe objects when they're created (or initialized). To do this, we create an `__init__` function (note the two underscore characters on each side, for a total of four). The `init` function sets the properties for an object when the object is first created. Python will automatically call this function when we create a new object. Here's how to use it.

```
>>> class Giraffe(Mammal):
        def __init__(self, spots):
            self.giraffe_spots = spots
```

First, we define the `__init__` function with the `self` and `spots` parameters. Just like the other functions we've defined in the class, the `__init__` function also needs to have `self` as the first parameter. Next, we set the `spots` parameter to an object variable (its property) called `giraffe_spots` by using the `self` parameter. You might think of this line of code as saying, "Take the value of the `spots` parameter and save it for later (using the `giraffe_spots` object variable)." Just as one function in a class can call another function using the `self` parameter, variables in the class are also accessed using `self` .

Next, if we create a couple of new giraffe objects (called `ozwald` and `gertrude`) and display their number of spots, you can see the initialization function in action:

```
>>> ozwald = Giraffe(100)
>>> gertrude = Giraffe(150)
>>> print(ozwald.giraffe_spots)
100
>>> print(gertrude.giraffe_spots)
150
```

First, we create an instance of the `Giraffe` class using the parameter value 100. This has the effect of calling the `__init__` function and using

100 for the value of the `spots` parameter. Next, we create another instance of the `Giraffe` class with a value of 150. Lastly, we print the object variable `giraffe_spots` for each of our giraffe objects, and we see that the results are 100 and 150. It worked!

Remember, when we create an object of a class, such as `ozwald` in this example, we can refer to its variables or functions using the dot operator and the name of the variable or function we want to use (for example, `ozwald.giraffe_spots`). But when we're creating functions inside a class, we refer to those same variables (and other functions) by using the `self` parameter (`self.giraffe_spots`).

WHAT YOU LEARNED

In this chapter, we used classes to create categories of things and made objects (or instances) of those classes. You learned how the child of a class inherits the functions of its parent, and that even though two objects are of the same class, they're not necessarily clones. For example, two giraffe objects can have their own distinct number of spots.

You learned how to call (or run) functions on an object and how object variables are a way of saving values in those objects. Lastly, we used the `self` parameter in functions to refer to other functions and variables. These concepts are fundamental to Python, and you'll see them multiple times throughout the rest of this book.

PROGRAMMING PUZZLES

Give the following examples a try to experiment with creating your own functions. The solutions can be found at *http://python-for-kids.com* .

#1: THE GIRAFFE SHUFFLE

Add functions to the `Giraffe` class to move the giraffe's left and right feet forward and backward. A function for moving the left foot forward might look like this:

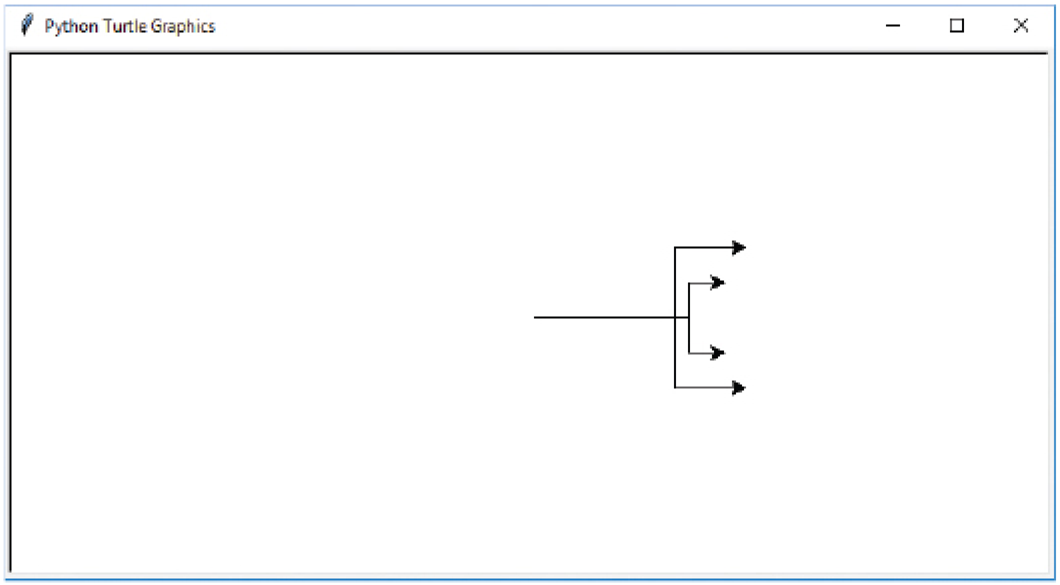
```
>>> def left_foot_forward(self):
        print('left foot forward')
```

Then create a function called `dance` to teach our giraffes to dance (the function will call the four foot functions you've just created). The result of calling this new function will be a simple dance:

```
>>> harriet = Giraffe()
>>> harriet.dance()
left foot forward
left foot back
right foot forward
right foot back
left foot back
right foot back
right foot forward
left foot forward
```

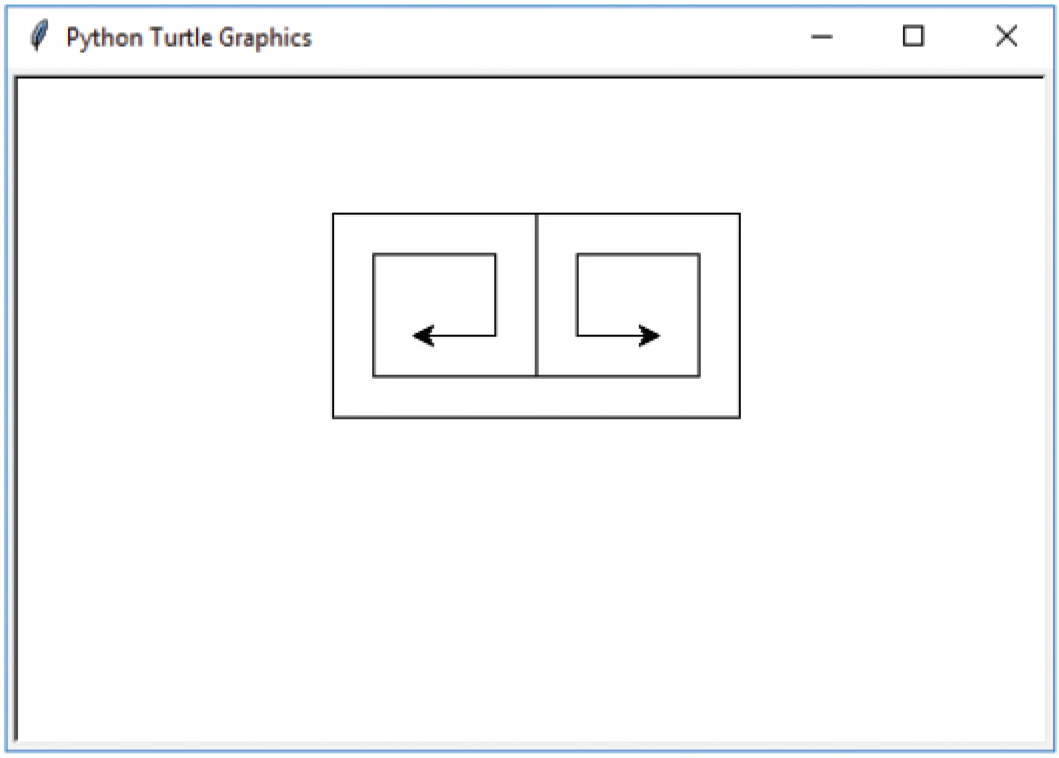
#2: TURTLE PITCHFORK

Create the following picture of a sideways pitchfork using four `Turtle` objects (the exact length of the lines isn't important). Remember to import the `turtle` module first!



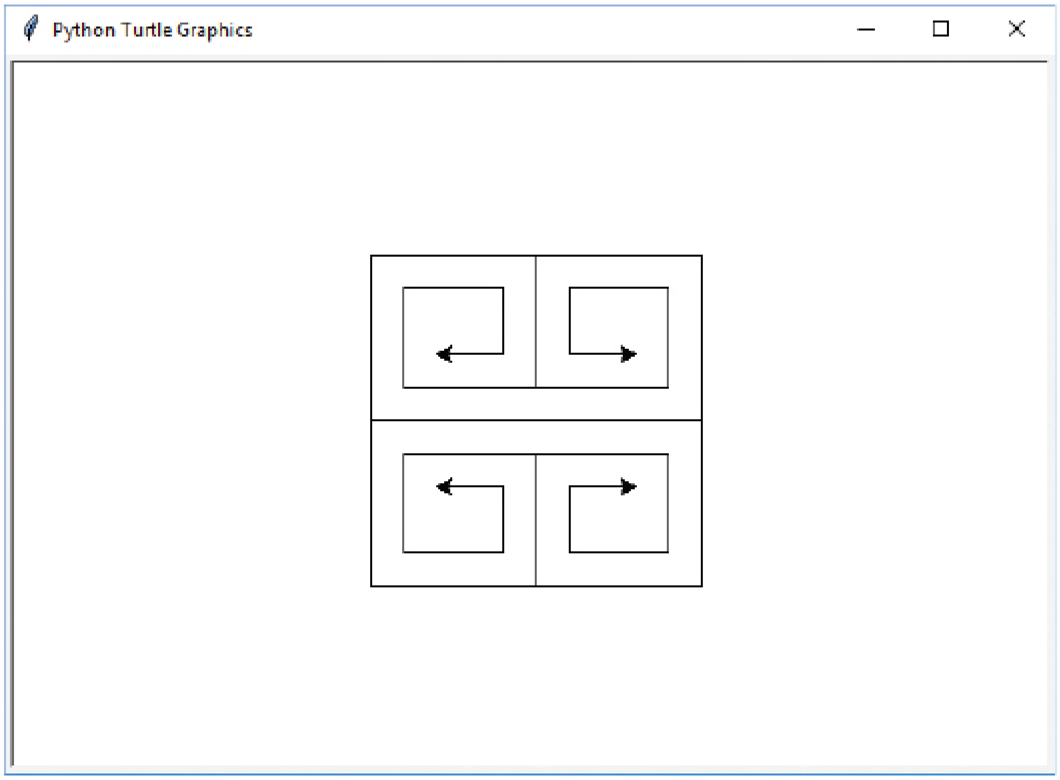
#3: TWO SMALL SPIRALS

Create the following picture of two small spirals using two `Turtle` objects (again, the exact size of the spirals isn't important).



#4: FOUR SMALL SPIRALS

Let's take the two spirals we created in the previous code and make a mirror image to create four spirals, which should look like the following image.


Python Turtle Graphics

9
MORE TURTLE GRAPHICS

Let's return to the turtle module we began using in Chapter 4 . In this chapter, we'll learn that Python turtles can do a lot more than draw plain black lines. You can use them to draw more advanced geometric shapes, create different colors, and even fill your shapes with color.

STARTING WITH THE BASIC SQUARE

We've previously used the turtle module to draw simple shapes. Let's import the turtle module and create the Turtle object:

```
>>> import turtle
>>> t = turtle.Turtle()
```

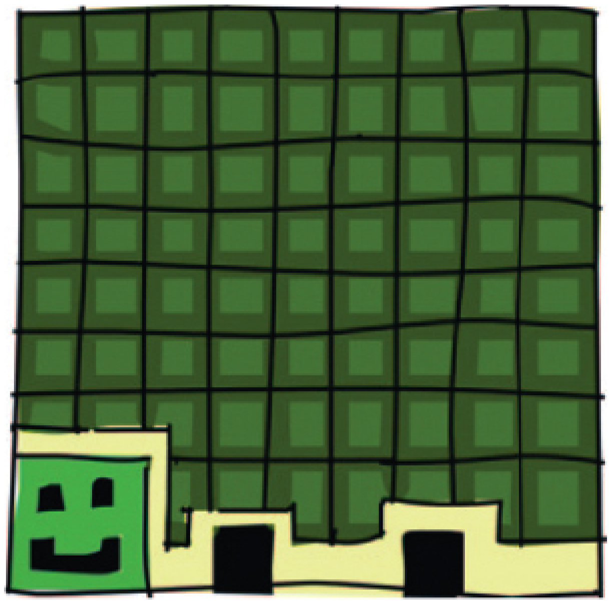
We used the following code in Chapter 4 to create a square:

```
>>> t.forward(50)
>>> t.left(90)
>>> t.forward(50)
>>> t.left(90)
>>> t.forward(50)
>>> t.left(90)
>>> t.forward(50)
>>> t.left(90)
```

In Chapter 6 , we learned about `for` loops. With our newfound knowledge, we can use a `for` loop to simplify this code, like so:

```
>>> t.reset()
>>> for x in range(1, 5):
        t.forward(50)
        t.left(90)
```

On the first line, we tell the `Turtle` object to reset itself. Next, we start a `for` loop that will count from 1 to 4 with `range(1, 5)` . With the following lines, in each run of the loop, we move forward 50 pixels and turn left 90 degrees. Because we've used a `for` loop, this code is a little shorter than the previous version—ignoring the `reset` line, we've gone from eight lines down to three.



DRAWING STARS

Now, with a few simple changes to our `for` loop, we can create something even more interesting. Enter the following:

```
>>> t.reset()
>>> for x in range(1, 9):
        t.forward(100)
        t.left(225)
```

This code produces an eight-point star, as in Figure 9-1 .

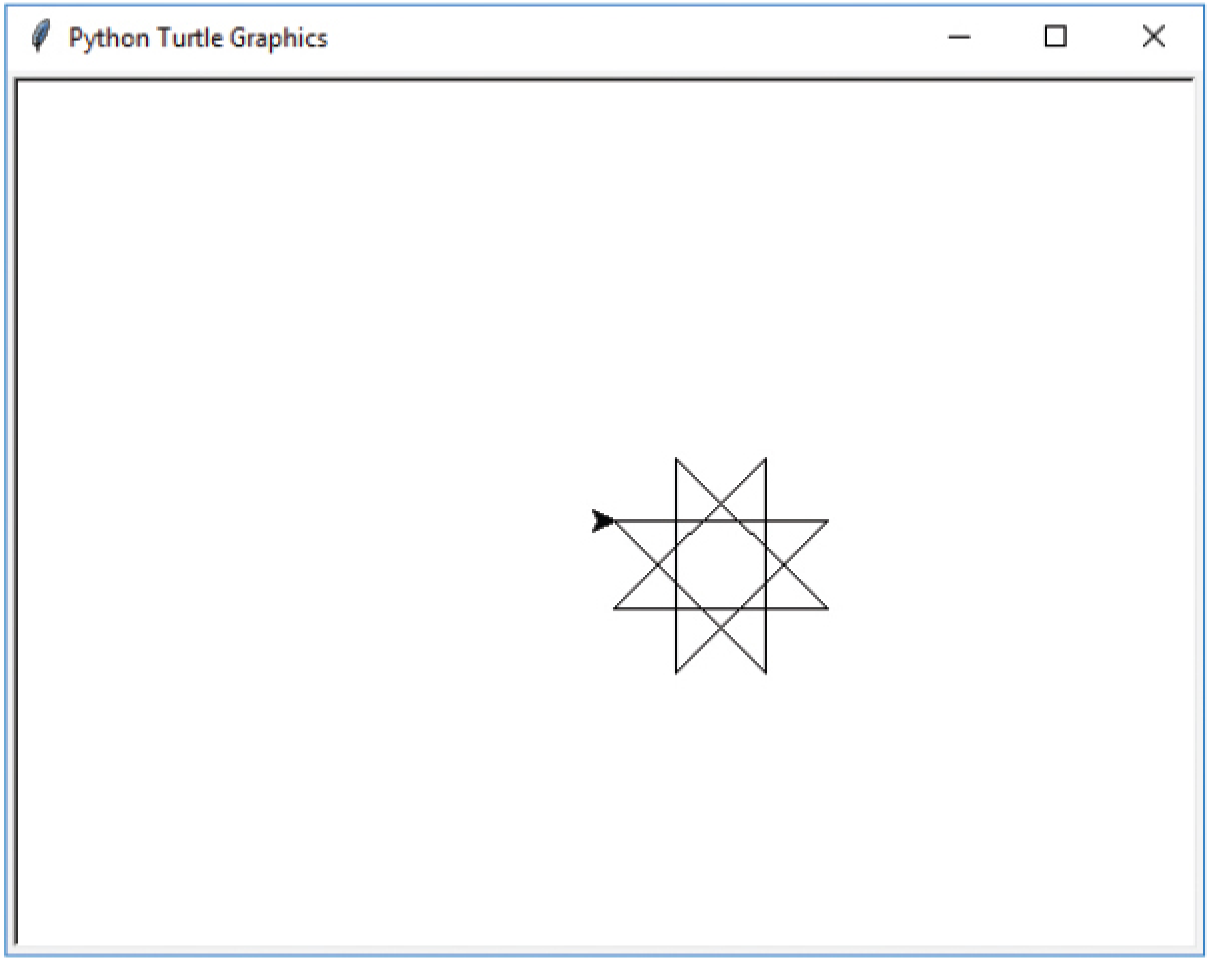


Figure 9-1: Eight-point star

The code itself is very similar to the code we used to draw a square, with a few exceptions:

- Rather than looping four times with `range(1, 5)` , we loop eight times with `range(1, 9)` .
- Rather than moving forward 50 pixels, we move 100 pixels.
- Rather than turning 90 degrees, we turn 225 degrees to the left.

Let's develop our star a bit more. By using a 175-degree angle and looping 37 times, we can make a star with even more points. Enter the following code:

```
>>> t.reset()
>>> for x in range(1, 38):
        t.forward(100)
        t.left(175)
```

You can see the result of running this code in Figure 9-2 .

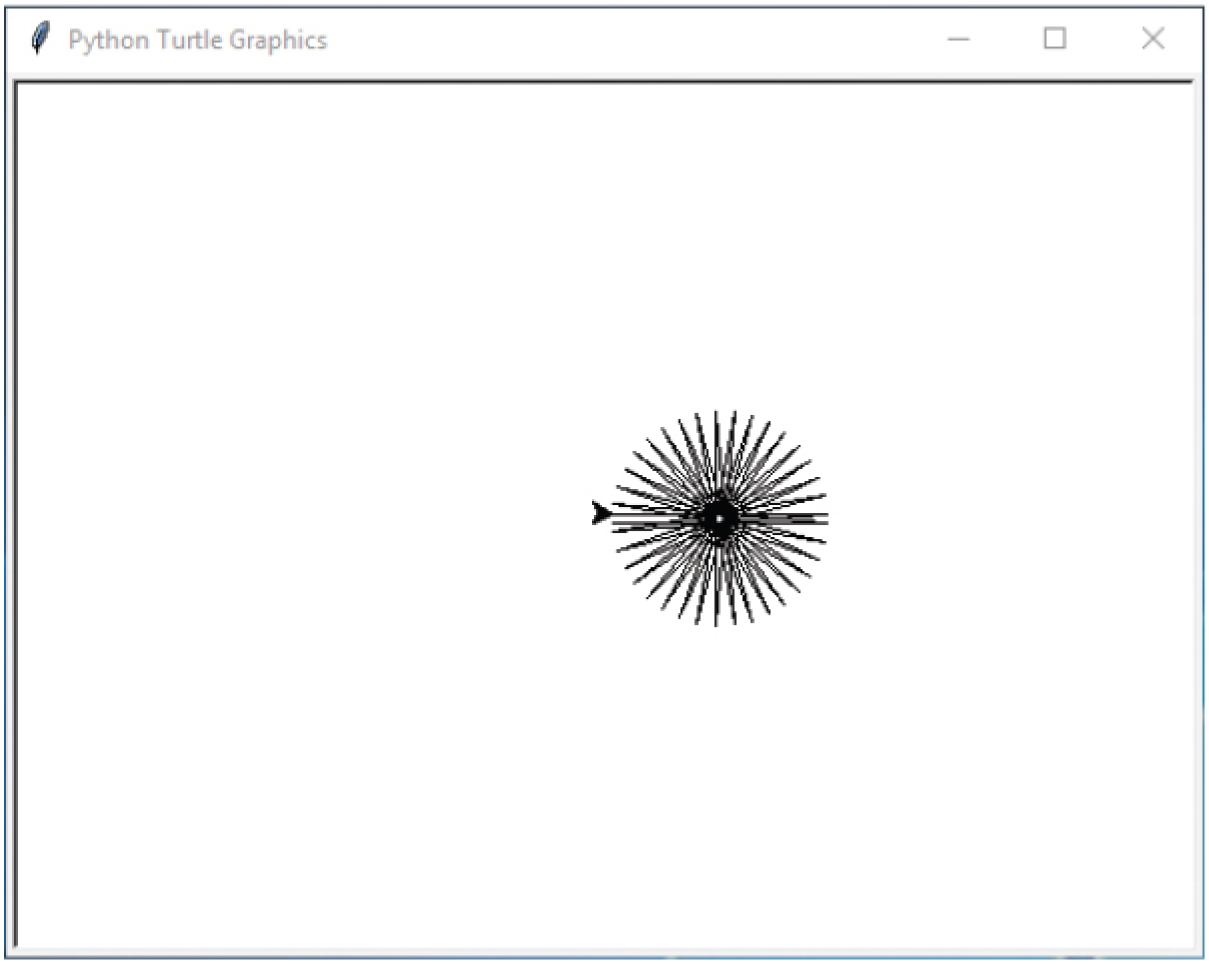


Figure 9-2: Multi-point star

Now, try entering this code to produce a spiraling star:

```
>>> t.reset()
>>> for x in range(1, 20):
        t.forward(100)
        t.left(95)
```

By changing the degree of the turn and reducing the number of loops, the turtle ends up drawing quite a different style of star, which you can see in Figure 9-3 .

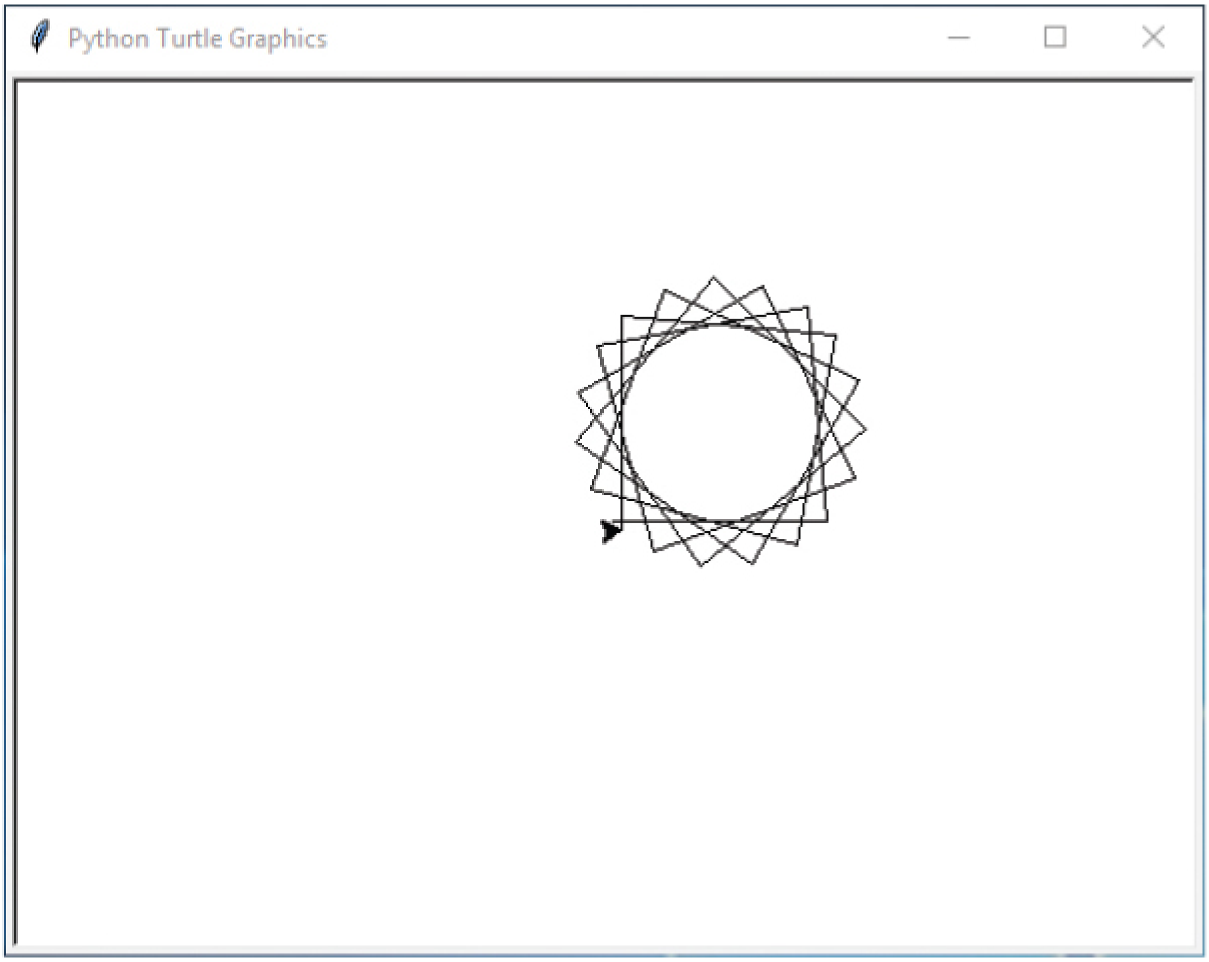
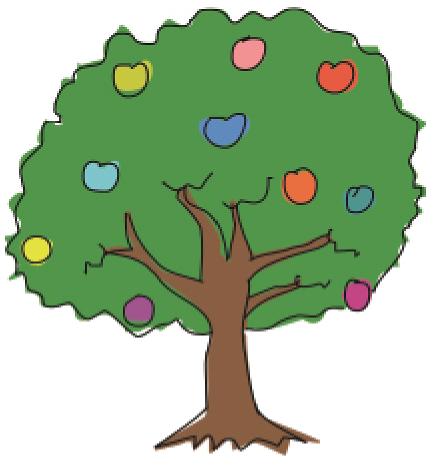


Figure 9-3: Spiral star

Using similar code, we can create a variety of shapes, from a basic square to a spiral star. As you can see, using `for` loops made it much simpler to draw these shapes. Without `for` loops, our code would have required a lot of tedious typing.



Let's try using an `if` statement to control how the turtle will turn and draw another star variation. In this example, we want the turtle to turn one angle the first time, and then another angle the next time:

```
   t.reset()
❶ for x in range(1, 19):
    ❷ t.forward(100)
       if x % 2 == 0:
           t.left(175)
       else:
           t.left(225)
```

Here, we create a loop that will run 18 times ❶ and tell the turtle to move forward 100 pixels ❷ . We've also added the `if` statement, which checks to see if the variable `x` contains an even number by using a *modulo operator* . The modulo operator is the `%` in the code `x % 2 == 0` , which is a way of saying, " *x* mod 2 is equal to 0."

The code `x % 2` asks, "What is the amount left over when you divide the number in variable `x` into two equal parts?" For example, if we were to divide 7 balls into two parts, we would get two groups of 3 balls (making a total of 6 balls), and the remainder (the amount left over) would be 1 ball, as shown in Figure 9-4 .

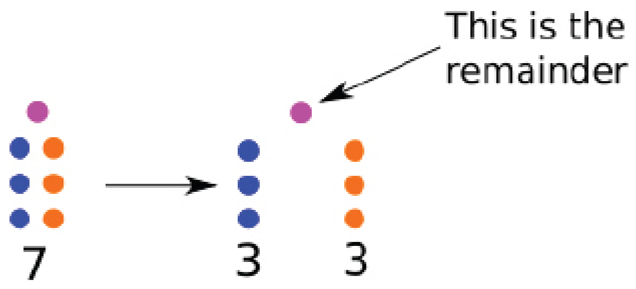


Figure 9-4: Dividing 7 balls into two equal parts

If we divided 13 balls into two parts, we would get two groups of 6 balls with 1 ball remaining (Figure 9-5).

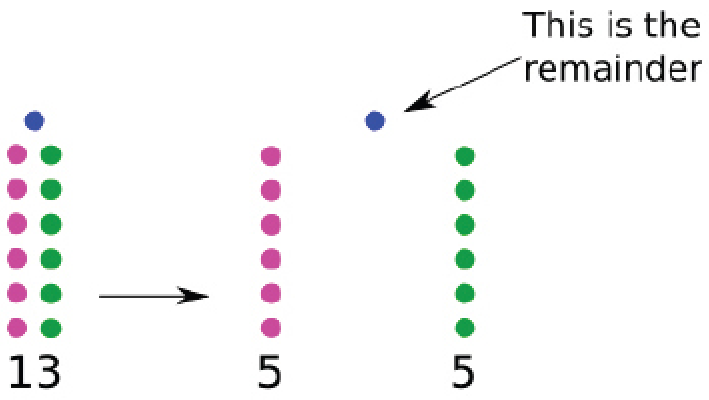


Figure 9-5: Dividing 13 balls into two equal parts

When we check to see if the remainder equals zero after dividing `x` by 2, we're actually asking whether it can be broken into two parts with no remainder. This method is a nice way to see if a number in a variable is even, because even numbers can always be divided into two equal parts.

On the fifth line of our code, we tell the turtle to turn left 175 degrees (`t.left(175)`) if the number in `x` is even (`if x % 2 == 0`); otherwise (`else`), on the final line, we tell it to turn 225 degrees (`t.left(225)`).

Figure 9-6 shows the result.

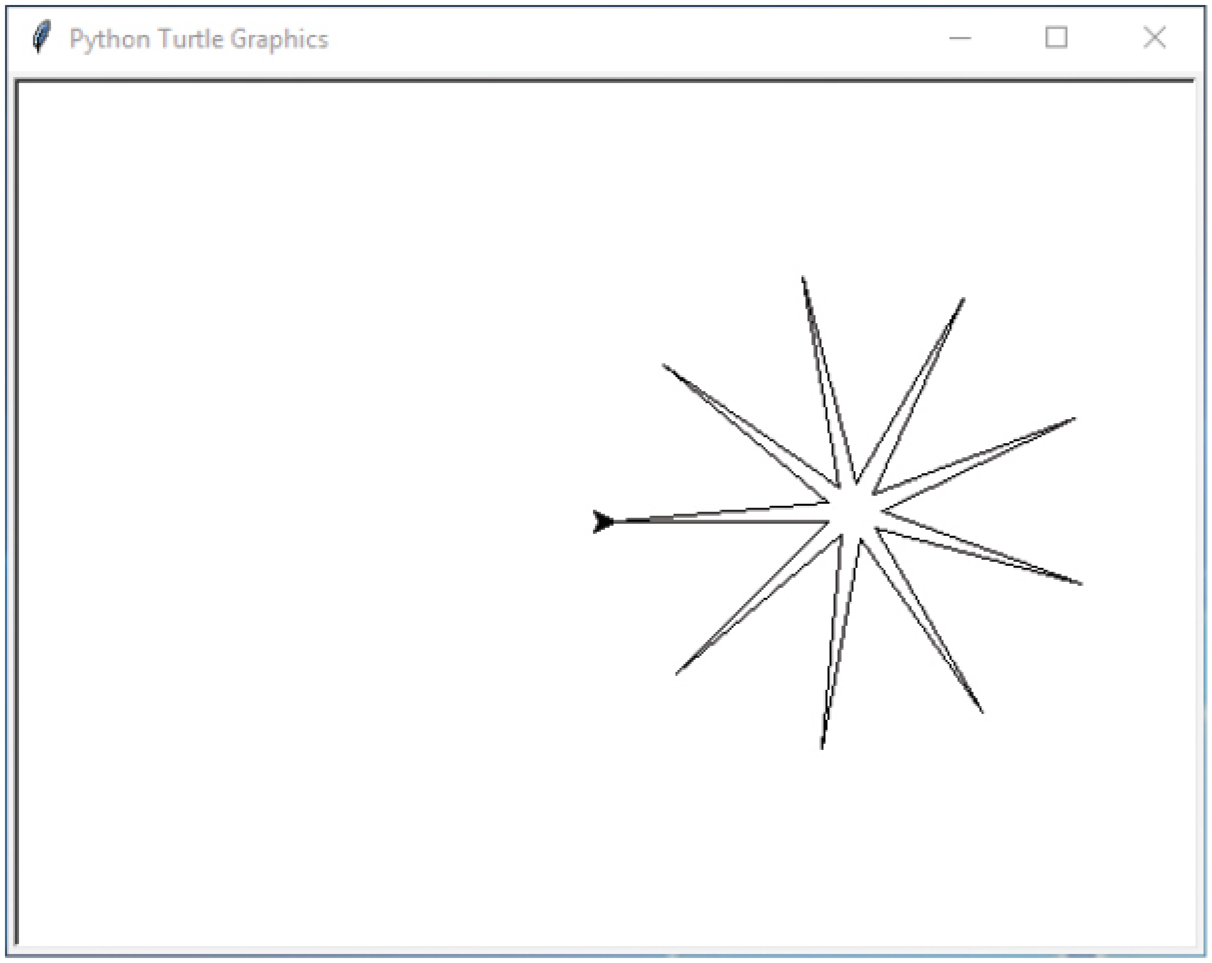


Figure 9-6: Nine-point star

If you tried the four spirals challenge in the previous chapter, you might have created four turtle objects and copied the code four times with slight differences for each turtle so that they draw the spiral in the correct direction. With `for` loops and `if` statements, you could do the same thing with much simpler code.

DRAWING A CAR

The turtle can also change colors and draw specific shapes. For this example, we'll draw a simple, if not silly-looking, car.

First, we'll draw the body of the car. In IDLE, select **File** ▸ **New File** , and then enter the following code in the window:

```
t.reset()
t.color(1,0,0)
t.begin_fill()
t.forward(100)
t.left(90)
t.forward(20)
t.left(90)
t.forward(20)
t.right(90)
t.forward(20)
t.left(90)
t.forward(60)
t.left(90)
t.forward(20)
t.right(90)
t.forward(20)
t.left(90)
t.forward(20)
t.end_fill()
```

Next, we'll draw the first wheel:

```
t.color(0,0,0)
t.up()
t.forward(10)
t.down()
t.begin_fill()
t.circle(10)
t.end_fill()
```

Lastly, we'll draw the second wheel:

```
t.setheading(0)
t.up()
t.forward(90)
t.right(90)
t.forward(10)
t.setheading(0)
t.begin_fill()
t.down()
t.circle(10)
t.end_fill()
```

Select **File** ‣ **Save As** and give the file a name, such as *car.py* . Select **Run** ‣ **Run Module** to try out the code. Our car can be seen in Figure 9-7 .

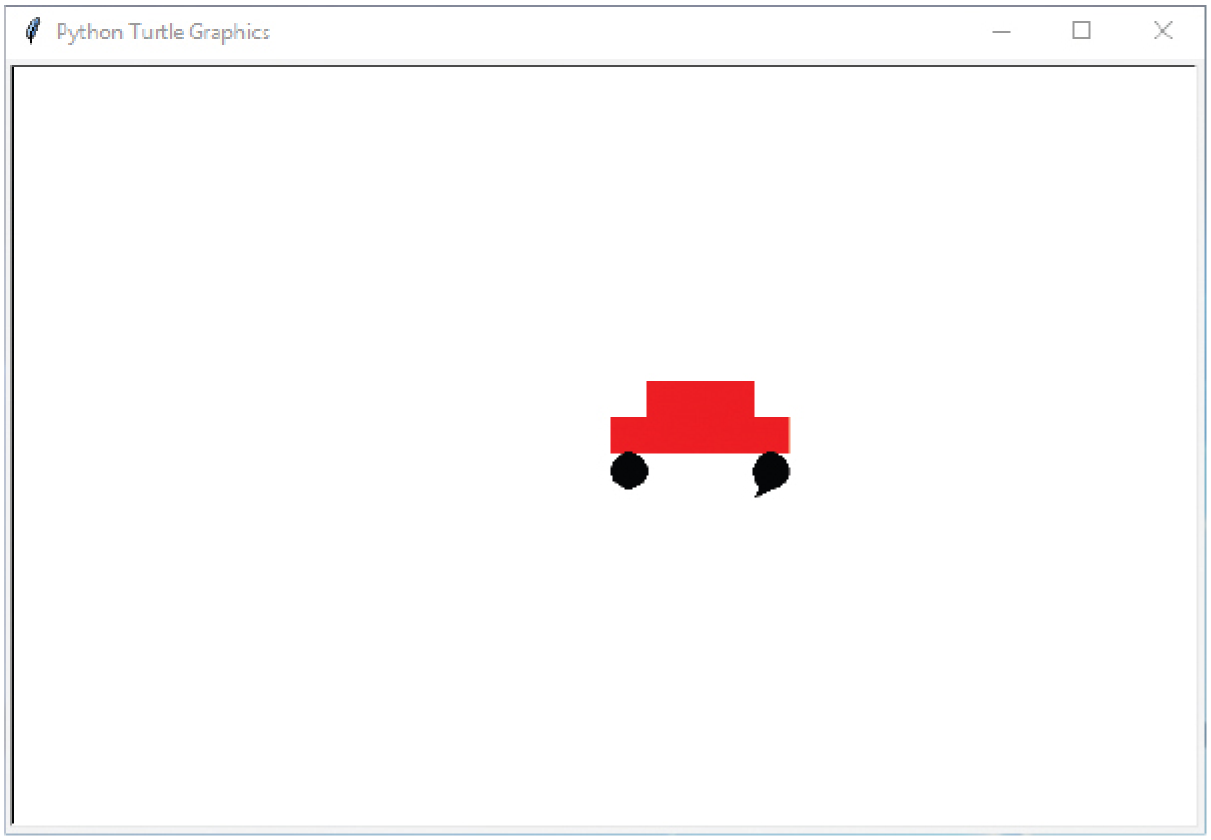


Figure 9-7: Turtle drawing a car

You may have noticed that a few new `turtle` functions have snuck into this code:

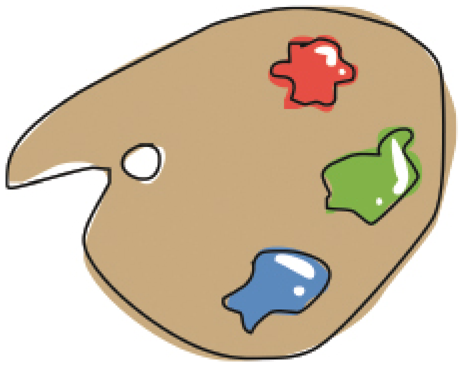
- `color` is used to change the color of the pen.
- `begin_fill` and `end_fill` are used to fill in an area of the canvas with a color.
- `circle` draws a circle of a particular size.
- `setheading` turns the turtle to face a particular direction.

Let's take a look at how we can use these functions to add color to our drawings.

COLORING THINGS IN

The `color` function takes three parameters. The first specifies the amount of red, the second the amount of green, and the third the amount of blue. For example, to get the bright red of the car, we used `color(1,0,0)` , which tells the turtle to use a 100 percent red.

This red, green, and blue color recipe is called *RGB* and is the way that colors are represented on your computer monitor. Mixing these primary colors produces other colors, just like when you mix blue and red paint to make purple, or yellow and red to make orange. The colors red, green, and blue are called *primary colors* because you cannot mix other shades to produce them.



Although we're not using paint to create colors on a computer monitor (we're using light), it may help to understand RGB by imagining you have three pots of paint: one red, one green, and one blue. Each pot is full, and we give each full pot a value of 1 (or 100 percent). We then mix all of the red paint and all of the green paint in a vat to produce yellow (that's 1 and 1 of each, or 100 percent of each color).

Now let's return to the world of code. To draw a yellow circle with the turtle, we would use 100 percent of both the red and green paint, but no blue, like this:

```
   >>> t.color(1,1,0)
   >>> t.begin_fill()
❶ >>> t.circle(50)
   >>> t.end_fill()
```

The `1,1,0` in the first line represents 100 percent red, 100 percent green, and 0 percent blue. On the next line, we tell the turtle to fill the shapes it draws with this RGB color, and then we tell it to draw a circle ❶ . The final line tells the turtle to fill the circle with the RGB color.

A FUNCTION TO DRAW A FILLED CIRCLE

To make it easier to experiment with colors, let's create a function from the code we used to draw a filled circle:

```
>>> def mycircle(red, green, blue):
        t.color(red, green, blue)
        t.begin_fill()
        t.circle(50)
        t.end_fill()
```

We can draw a bright green circle by using only the green paint, like so:

```
>>> mycircle(0, 1, 0)
```

Or we can draw a darker green circle by using only half the green paint (`0.5`):

```
>>> mycircle(0, 0.5, 0)
```

To play with the RGB colors on your screen, try drawing a circle first with full red, then half red (`1` and `0.5`), and with full blue, then half blue, like this:

```
>>> mycircle(1, 0, 0)
>>> mycircle(0.5, 0, 0)
>>> mycircle(0, 0, 1)
>>> mycircle(0, 0, 0.5)
```

NOTE

If your canvas starts to get cluttered, use `t.reset()` to delete old drawings. Also remember that you can move the turtle without drawing lines by using *`t.up()` to lift the pen and `t.down()` to set it back down again.*

Various combinations of red, green, and blue will produce a huge variety of colors, like gold:

```
>>> mycircle(0.9, 0.75, 0)
```

Or light pink:

```
>>> mycircle(1, 0.7, 0.75)
```

And here are two different shades of orange:

```
>>> mycircle(1, 0.5, 0)
>>> mycircle(0.9, 0.5, 0.15)
```

Try mixing some colors yourself!

CREATING PURE BLACK AND WHITE

What happens when you turn off all the lights at night? Everything goes black. The same thing happens with colors on a computer. No light means no color, so a circle with 0 for all of the primary colors creates black:

```
>>> mycircle(0, 0, 0)
```

Figure 9-8 shows the result.

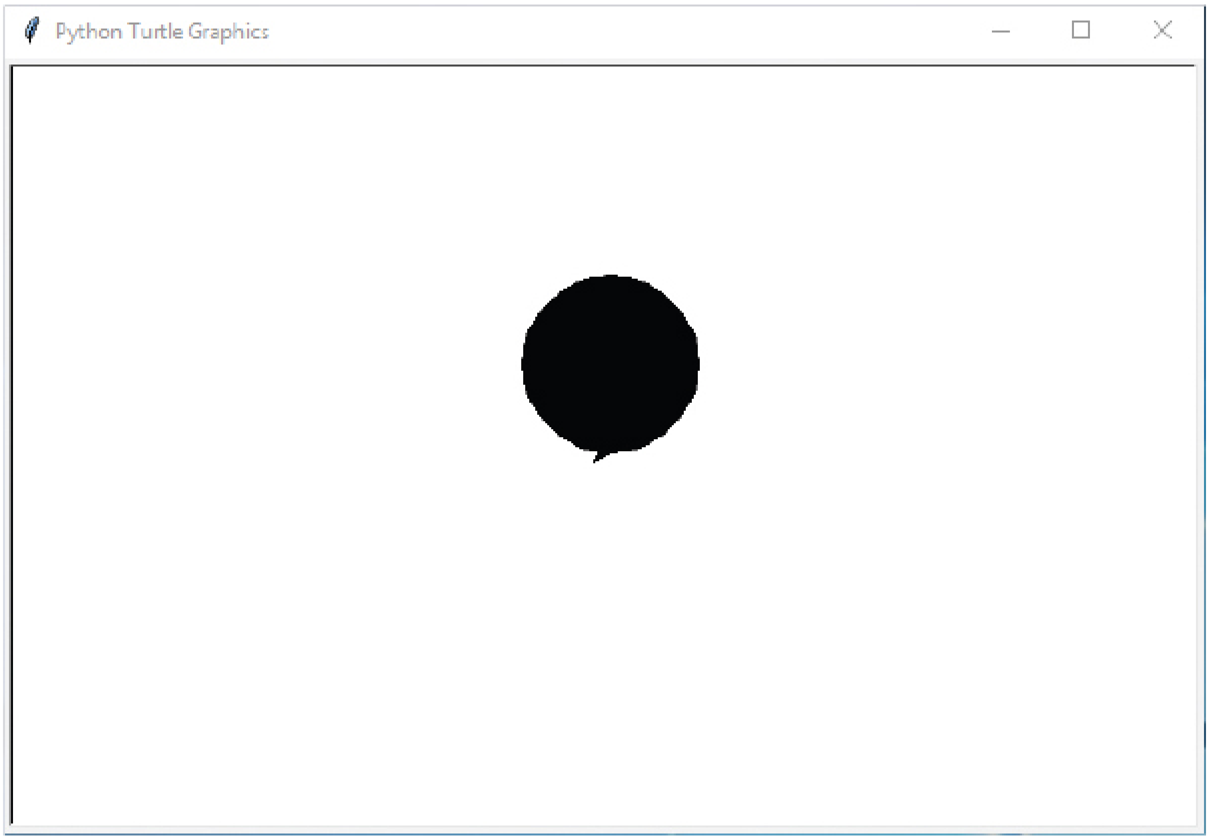


Figure 9-8: Black circle

If you use 100 percent of all three colors, you get white. Enter the following to wipe out your black circle:

```
>>> mycircle(1, 1, 1)
```

A SQUARE-DRAWING FUNCTION

Now we'll try a few more experiments with shapes. Let's use the square-drawing function from the beginning of the chapter and pass it the size of the square as a parameter:

```
>>> def mysquare(size):
        for x in range(1, 5):
            t.forward(size)
            t.left(90)
```

Test your function by calling it with size 50, like so:

```
>>> mysquare(50)
```

This produces the small square in Figure 9-9 .

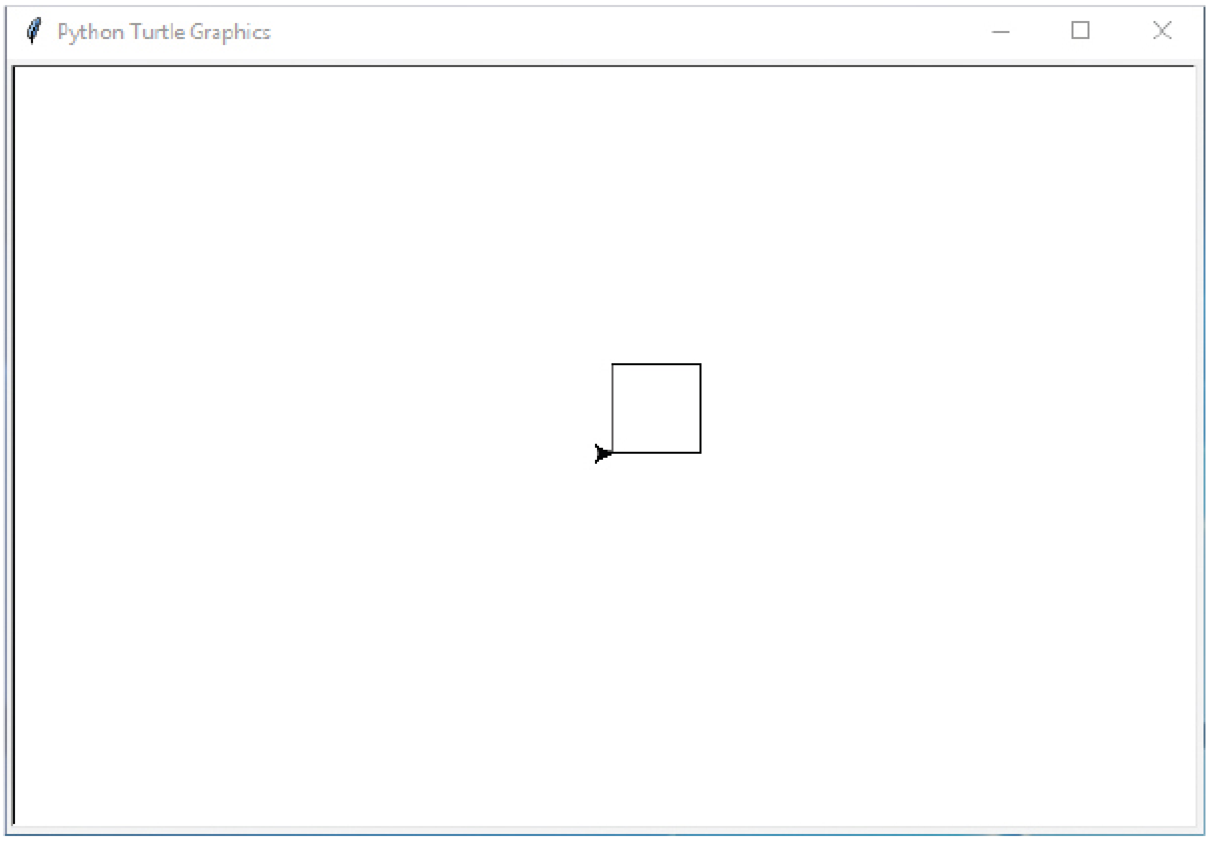


Figure 9-9: Turtle drawing a small square

Now let's try our function with different sizes. The following code creates five consecutive squares of size 25, 50, 75, 100, and 125:

```
>>> t.reset()
>>> mysquare(25)
>>> mysquare(50)
>>> mysquare(75)
>>> mysquare(100)
>>> mysquare(125)
```

These squares should look like Figure 9-10 .

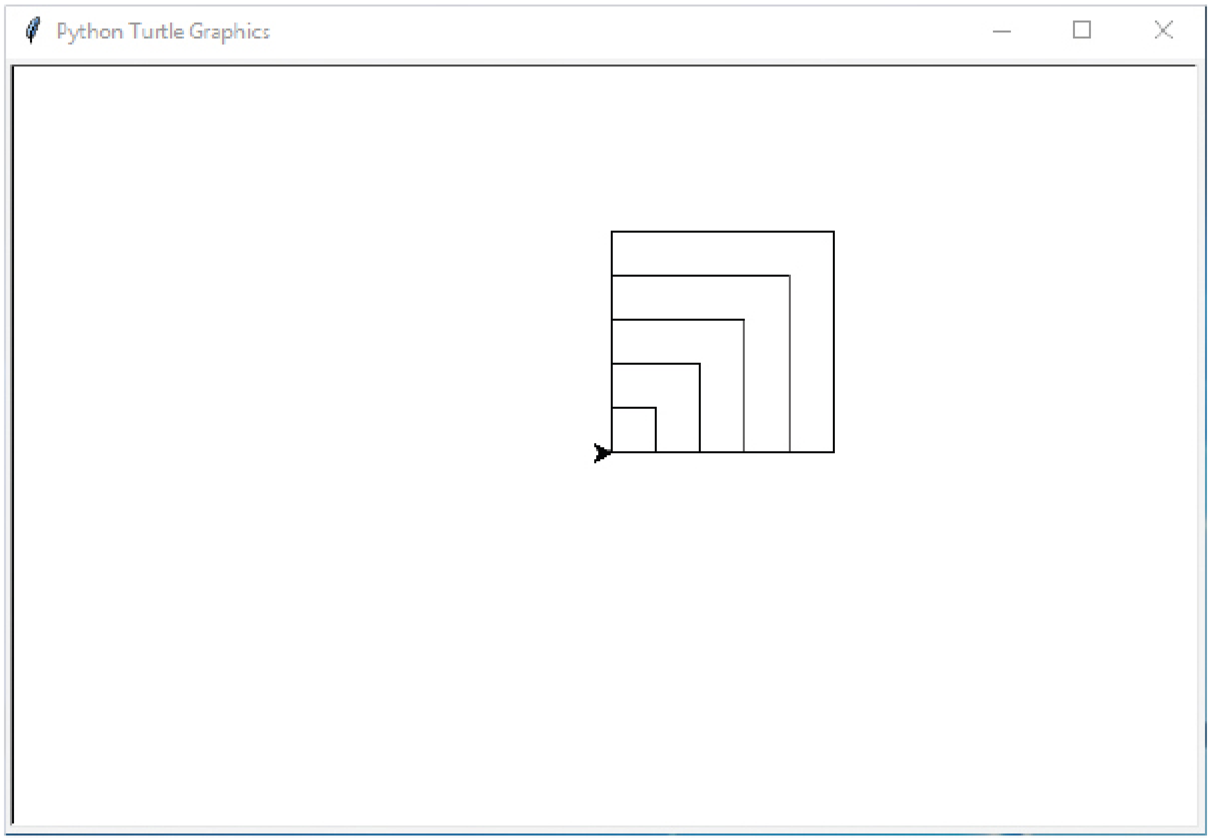


Figure 9-10: Turtle drawing multiple squares

DRAWING FILLED SQUARES

To draw a filled square, we need to reset the canvas, begin filling, and then call the square function again, with this code:

```
>>> t.reset()
>>> t.begin_fill()
>>> mysquare(50)
```

You should see an empty square until you end filling:

```
>>> t.end_fill()
```

Your square should look like Figure 9-11 .

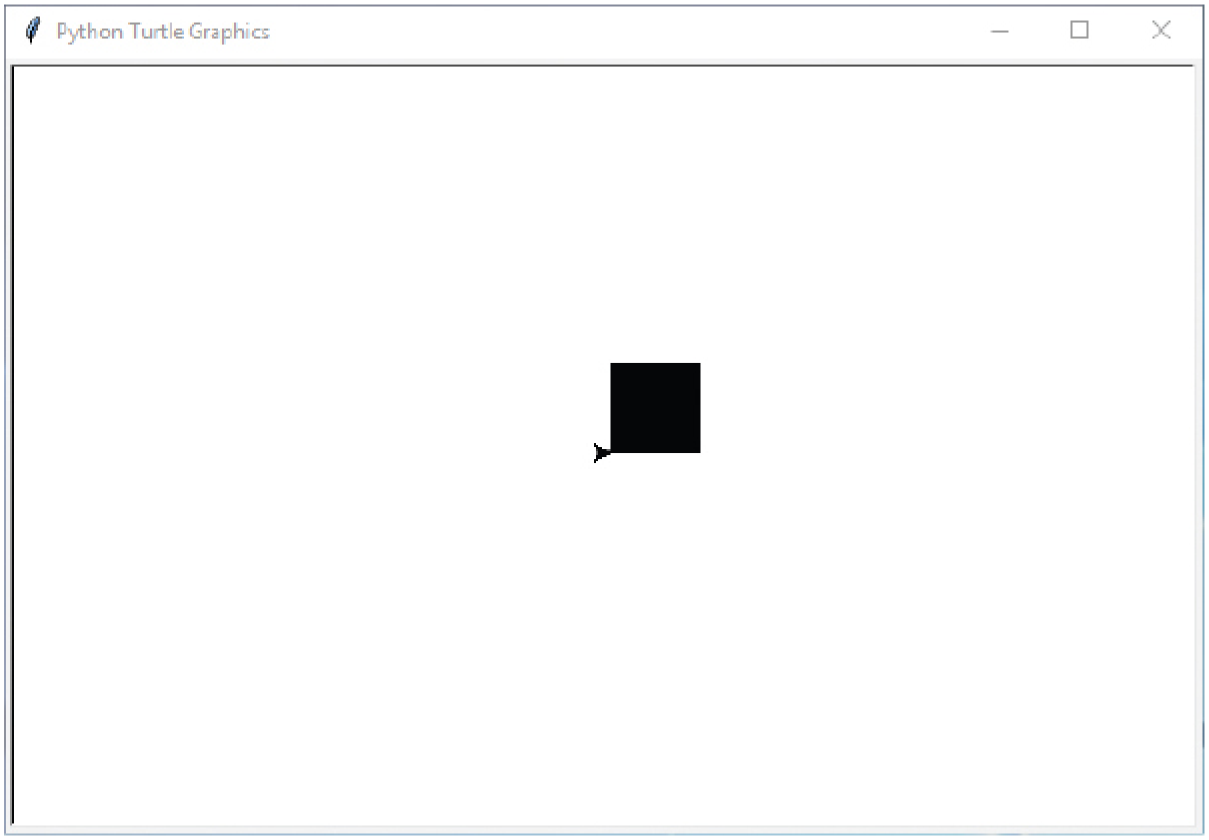


Figure 9-11: Turtle drawing a filled square

Let's change this function so we can draw either a filled or an unfilled square. To do so, we need another parameter and slightly more complicated code:

```
>>> def mysquare(size, filled):
        if filled == True:
            t.begin_fill()
        for x in range(1, 5):
            t.forward(size)
            t.left(90)
        if filled == True:
            t.end_fill()
```

On the first line, we change the definition of our function to take two parameters: `size` and `filled` . Next, we check to see whether the value of filled is set to `True` with `if filled == True` . If it is, we call `begin_fill` to tell the turtle to fill the shape we drew. We then loop four times (`for x in range(1, 5)`) to draw the four sides of the square (moving forward and left) before checking again to see whether filled is `True` . If it is, we turn filling off with `t.end_fill` , and the turtle fills the square with color.

Now we can draw a filled square with this line:

```
>>> mysquare(50, True)
```

Or we can create an unfilled square with this line:

```
>>> mysquare(150, False)
```

After these two calls to the `mysquare` function, we get Figure 9-12 , which looks a bit like a square eye.

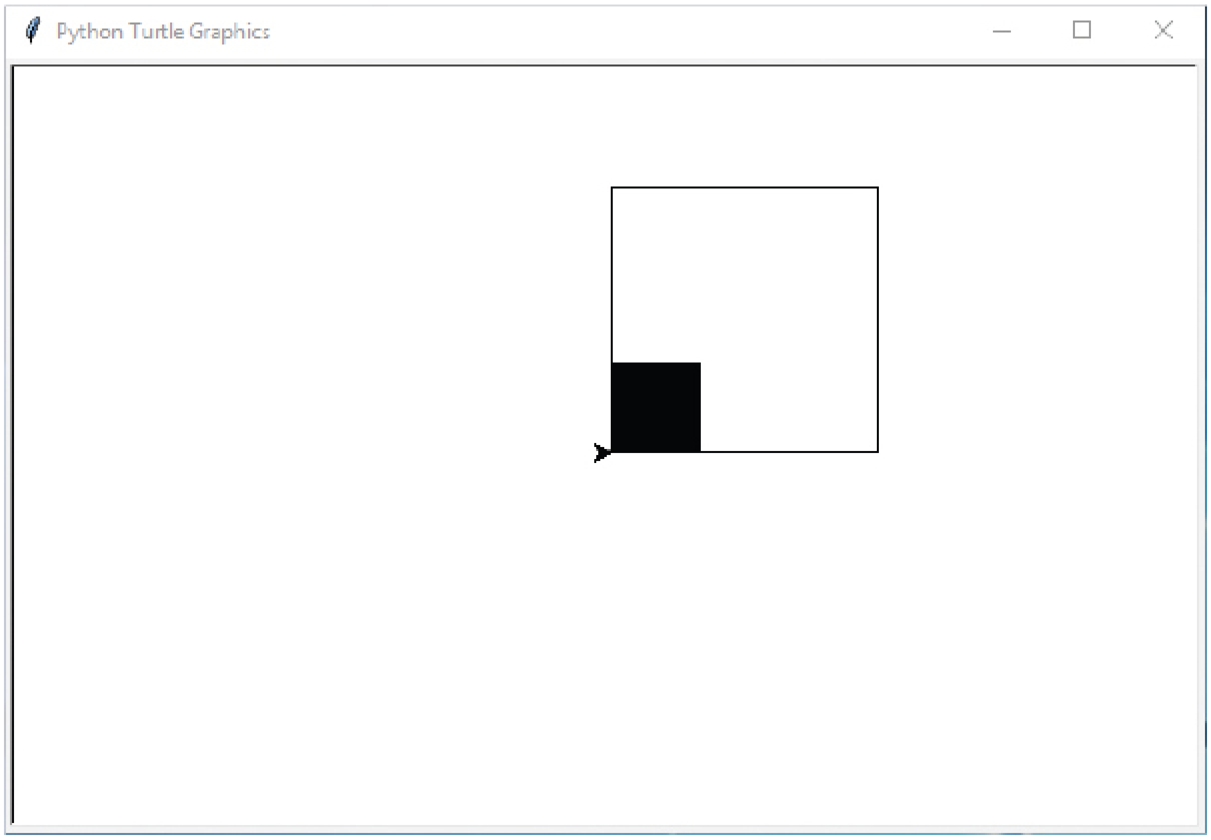


Figure 9-12: Turtle drawing a square eye

But there's no sense in stopping here. You can draw all sorts of shapes and fill them with color.

DRAWING FILLED STARS

For our final example, we'll add color to the star we drew earlier. The original code looked like this:

```
for x in range(1, 19):
    t.forward(100)
    if x % 2 == 0:
        t.left(175)
    else:
        t.left(225)
```

Now we'll make a `mystar` function. We'll use the `if` statements from the `mysquare` function and add the `size` parameter:

```
>>> def mystar(size, filled):
        if filled == True:
            t.begin_fill()
        for x in range(1, 19):
            t.forward(size)
            if x % 2 == 0:
                t.left(175)
            else:
                t.left(225)
        if filled == True:
            t.end_fill()
```

In the first two lines of this function, we check to see if `filled` is `True` ; if it is, we begin filling. We check again in the last two lines, and if filled is `True` , we stop filling. Also, as with the `mysquare` function, we pass the size of the star in the parameter `size` , and use that value when we call `t.forward` .

Now let's set the color to gold (90 percent red, 75 percent green, and 0 percent blue), and then call the function again:

```
>>> t.color(0.9, 0.75, 0)
>>> mystar(120, True)
```

The turtle will draw the filled star in Figure 9-13 .

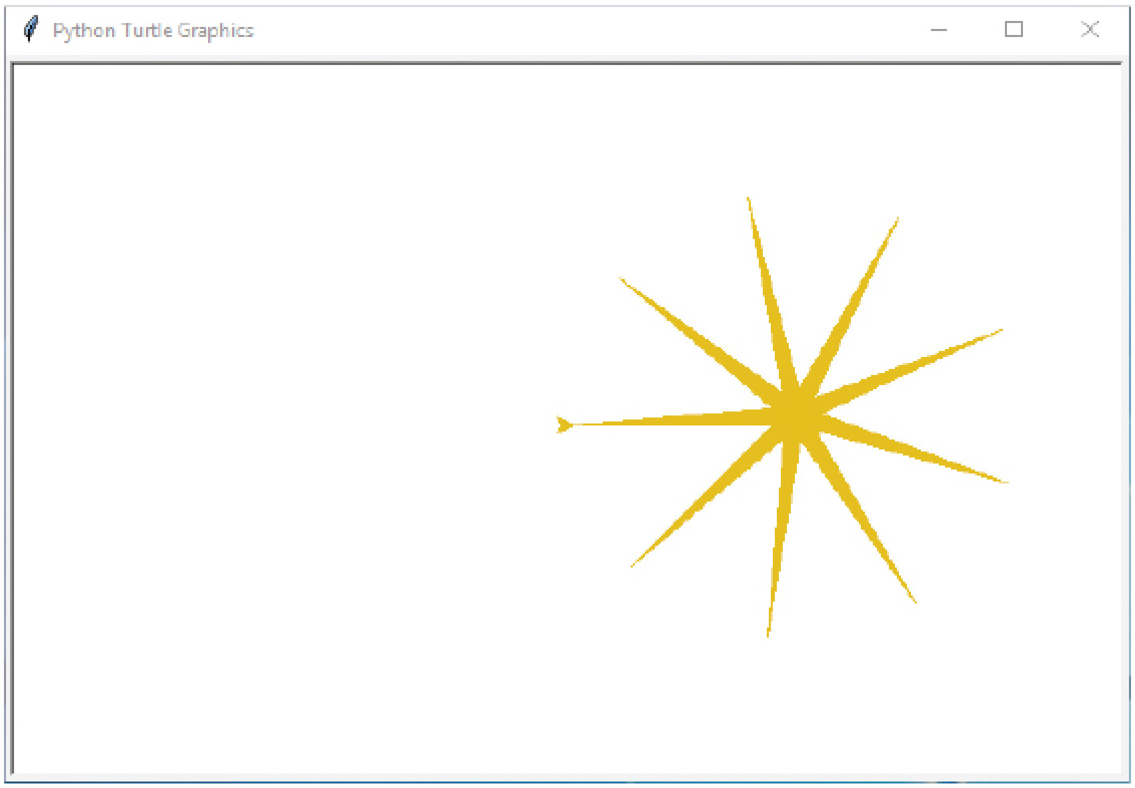


Figure 9-13: Drawing a gold star

To add an outline to the star, change the color to black and redraw the star without filling:

```
>>> t.color(0,0,0)
>>> mystar(120, False)
```

Now the star is gold with a black outline, like Figure 9-14 .

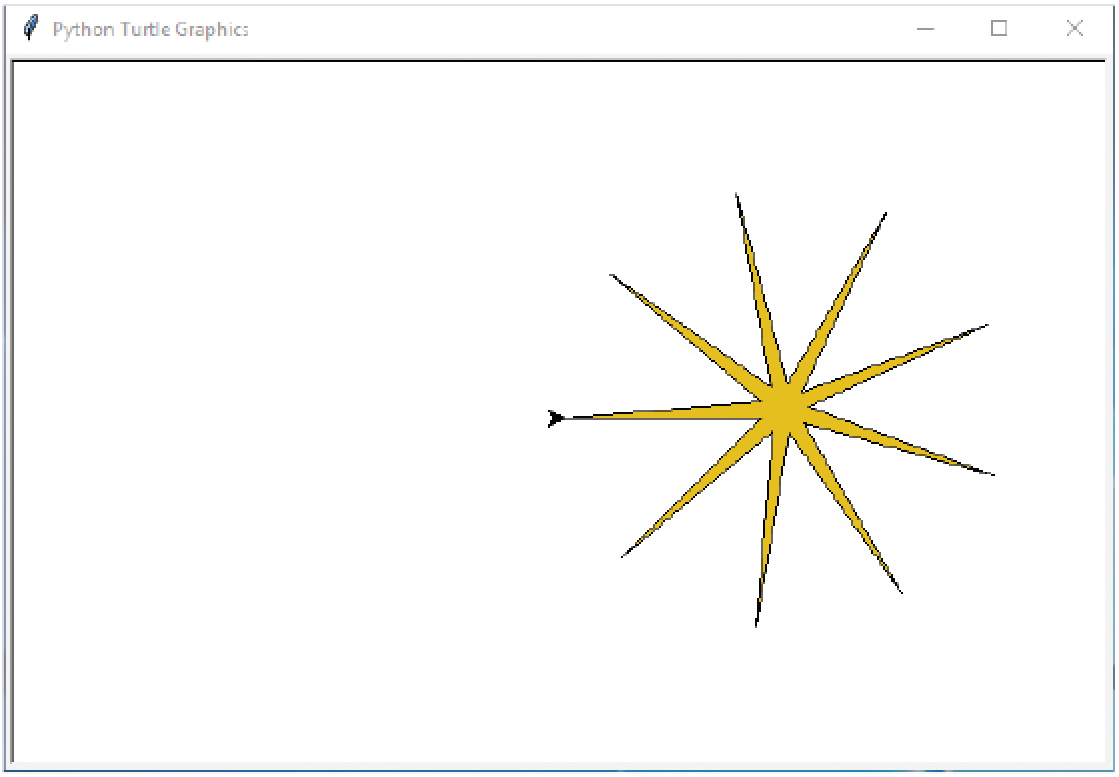


Figure 9-14: Drawing a gold star with an outline

WHAT YOU LEARNED

In this chapter, you learned how to use the `turtle` module to draw geometric shapes, using `for` loops and `if` statements to control what the turtle does on the screen. We changed the color of the turtle's line and filled the shapes that it drew. We also reused the drawing code in some functions to make it easier to draw shapes with different colors with a single call to a function.

PROGRAMMING PUZZLES

In the following experiments, you'll draw your own shapes with the turtle. As always, the solutions can be found at *http://python-for-kids.com*
.

#1: DRAWING AN OCTAGON

We've drawn stars, squares, and rectangles in this chapter. How about creating a function to draw an eight-sided shape like an octagon? (Hint: Try turning the turtle 45 degrees.) Your shape should look similar to Figure 9-15 .

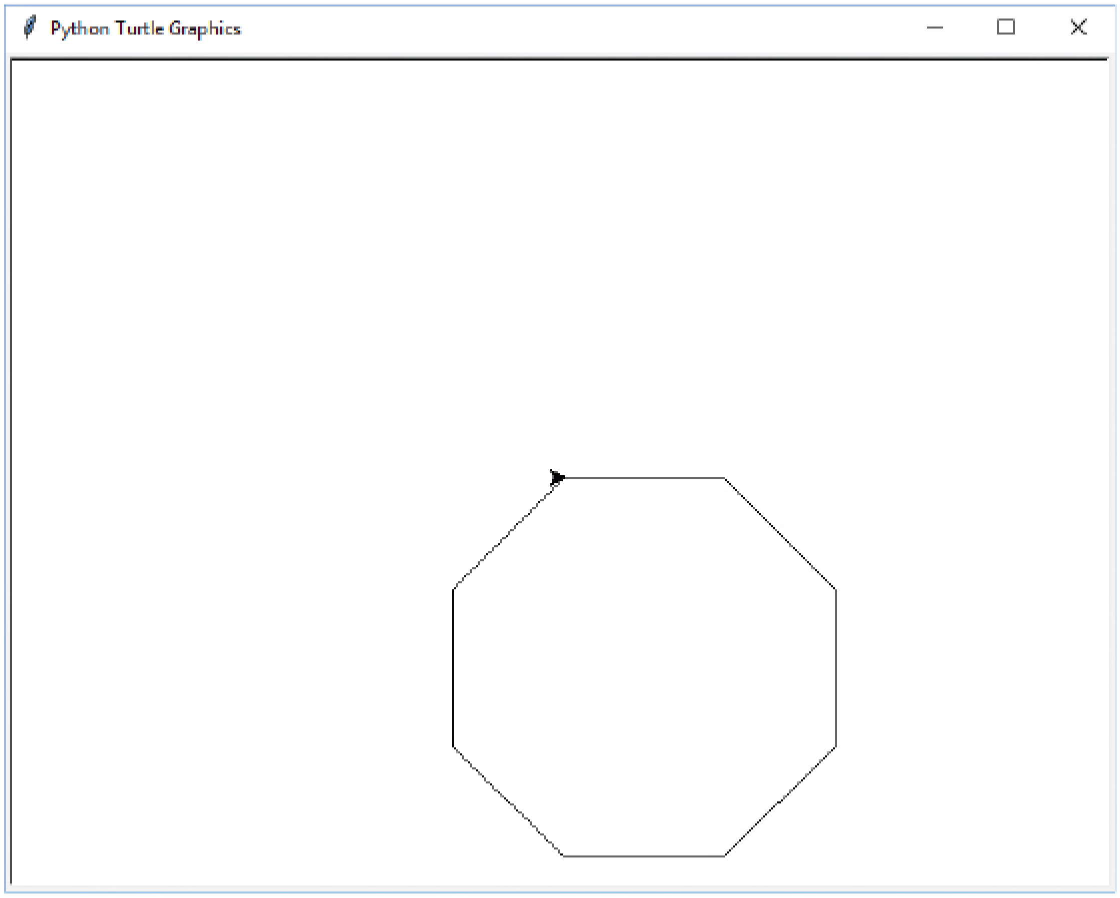


Figure 9-15: Drawing an octagon

#2: DRAWING A FILLED OCTAGON

Now that you have a function to draw an octagon, modify it to draw a filled octagon. Try drawing an octagon with an outline, as we did with the star. It should look similar to Figure 9-16 .

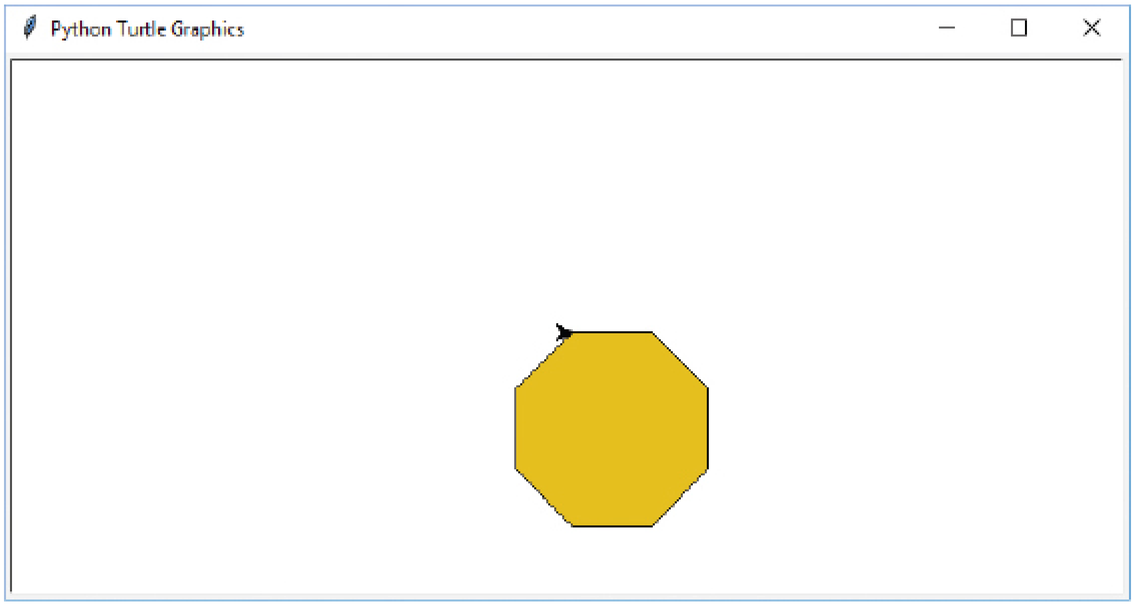


Figure 9-16: Drawing a filled octagon

#3: ANOTHER STAR-DRAWING FUNCTION

Create a function to draw a star that will take two parameters: the size and number of points. The beginning of the function should look something like this:

```
def draw_star(size, points):
```

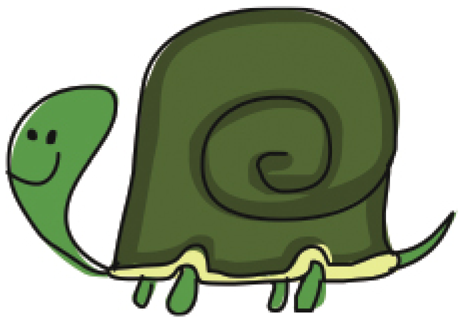
#4: FOUR SPIRALS REVISITED

Take the code you created for Programming Puzzle #4 in the previous chapter (to create four spirals) and draw the same spirals again—only this time, try using `for` loops and `if` statements to simplify your code.

10
USING TKINTER FOR BETTER GRAPHICS

The problem with using a turtle to draw is . . . that . . . turtles . . . are . . . really . . . slow. Even when a turtle is going at top speed, it's still not very fast. While this isn't really an issue for turtles, it is for computer graphics.

Computer graphics usually need to move fast. If you play games on a game console or computer, think for a moment about the graphics you see on the screen. Two-dimensional (2D) graphics are flat: the characters generally move only up and down or left and right, as in many Nintendo and phone games. In pseudo-three-dimensional (3D) games—ones that are almost 3D—images look a little more real, but the characters generally move only in relation to a flat plane (this is also known as *isometric graphics*). Finally, we have 3D games, with graphics that attempt to mimic reality. Whether games use 2D, pseudo-3D, or 3D graphics, all have one thing in common: the need to draw on the computer screen very quickly.



If you've never tried to create your own animation, try this simple project:

1. Get a blank pad of paper, and in the bottom corner of the first page, draw something (perhaps a stick figure).
2. On the corner of the next page, draw the same stick figure, but move its leg slightly.
3. On the next page, draw the stick figure again, with the leg moved a little more.
4. Gradually go through each page, drawing a modified stick figure on the bottom corner.

When you're finished, flip quickly through the pages, and you should see your stick figure moving. This is the basic method used with all animation, whether cartoons or video games. An image is drawn, and then drawn again with a slight change to create the illusion of movement. To make an image look like it is moving, you need to display each *frame* —or piece of the animation—very quickly.

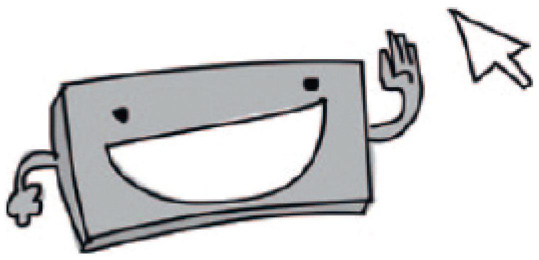
Python offers different ways to create graphics. In addition to the `turtle` module, you can use *external* modules (which you need to install separately), as well as the `tkinter` module, which you should already have in your standard Python installation. The `tkinter` module can be used to create full applications, like a simple word processor, as well as drawings. In this chapter, we'll explore using `tkinter` to create graphics.

CREATING A CLICKABLE BUTTON

For our first example, we'll use the tkinter module to create a basic application with a button. Enter this code:

```
>>> from tkinter import *
>>> tk = Tk()
>>> btn = Button(tk, text='click me')
>>> btn.pack()
```

On the first line, we import the contents of the tkinter module. The from *module-name* import * line allows us to use the contents of a module without using its name. In contrast, when using import turtle in previous examples, we needed to include the module name to access its contents, like so:



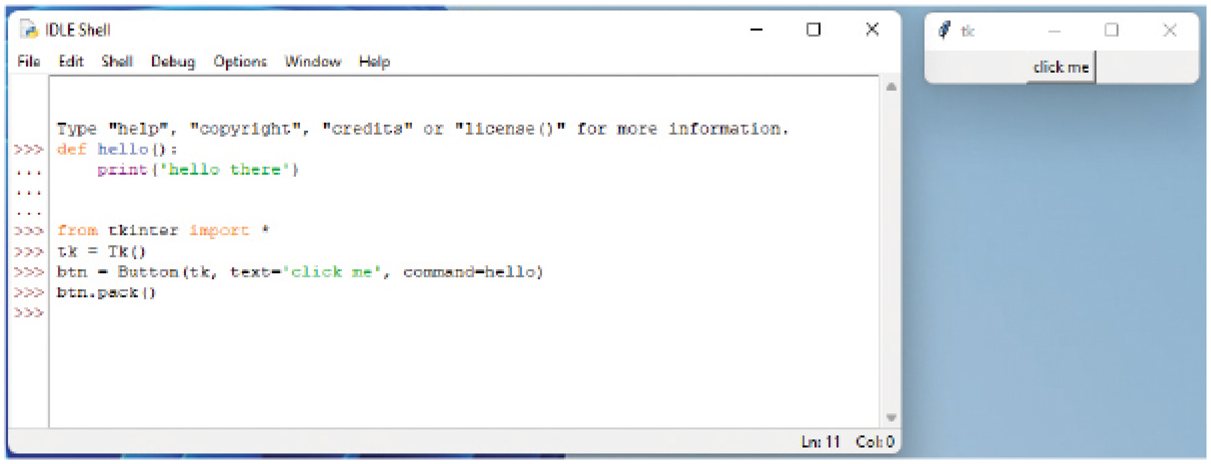
```
import turtle
t = turtle.Turtle()
```

When we use import * , we don't need to call turtle.Turtle , as we did in Chapters 4 and 9 . This is most useful when you're using modules with a lot of classes and functions, because it reduces the amount you need to type:

```
from turtle import *
t = Turtle()
```

On the next line in our button example, we create a variable containing an object of the class Tk with tk = Tk() , just like we create a Turtle object for the turtle. The tk object creates a basic window to which we can then add other things, such as buttons, input boxes, or a canvas to draw on. This is the main class provided by the tkinter module; without creating an object of the `Tk` class, you won't be able to do any graphics or animations.

On the third line, we create a button with `btn = Button` , and pass the `tk` variable as the first parameter and "click me" as the text the button will display with `text='click me'` . Although we've added this button to the window, it won't be displayed until you enter the line `btn.pack()` , which tells the button to appear. It also lines everything up correctly on the screen if there are other buttons or objects to display. The result should be something like Figure 10-1 .



```
Type "help", "copyright", "credits" or "license()" for more information.
>>> def hello():
...     print('hello there')
...
...
>>> from tkinter import *
>>> tk = Tk()
>>> btn = Button(tk, text='click me', command=hello)
>>> btn.pack()
>>>
```

Figure 10-1: A `tkinter` application with a single button

Right now, the Click Me button doesn't do much. You can click it all day, but nothing will happen until we change the code a bit. (Be sure to close the window you created earlier!)

First, we create a function to print some text:

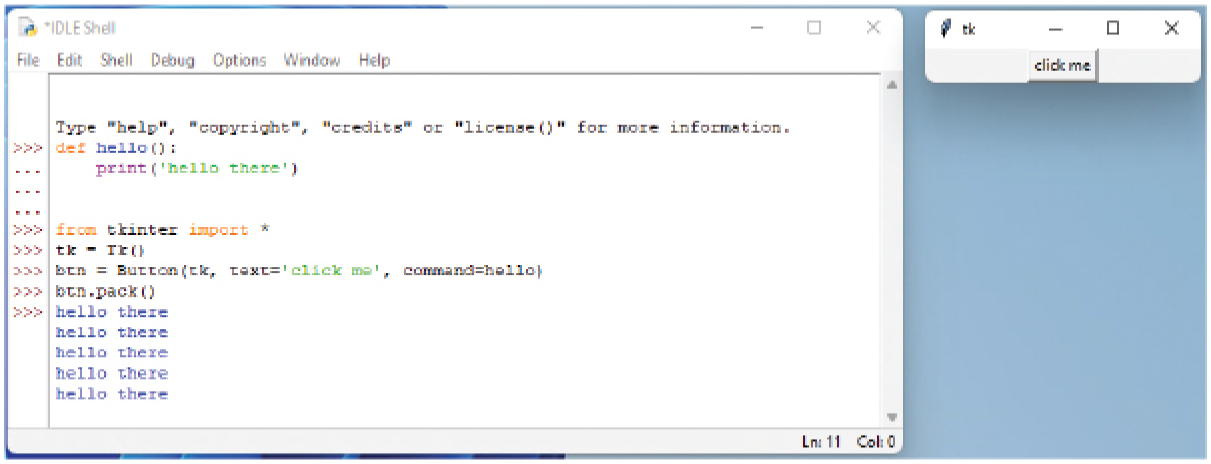
```
>>> def hello():
        print('hello there')
```

Then we modify our example to use this new function:

```
>>> from tkinter import *
>>> tk = Tk()
>>> btn = Button(tk, text='click me', command=hello)
>>> btn.pack()
```

We've made only a slight change to the previous version of this code, adding the `command` parameter, which tells Python to use the `hello` function when the button is clicked.

Now when you click the button, you'll see "hello there" written to the Python Shell. This will appear each time the button is clicked. In Figure 10-2 , I've clicked the button five times.



```
Type "help", "copyright", "credits" or "license()" for more information.
>>> def hello():
...     print('hello there')
...
...
>>> from tkinter import *
>>> tk = Tk()
>>> btn = Button(tk, text='click me', command=hello)
>>> btn.pack()
>>> hello there
hello there
hello there
hello there
hello there
```

Figure 10-2: Clicking the button

This is the first time we've used named parameters in any of our code examples, so let's talk about them a bit before continuing with our drawing.

USING NAMED PARAMETERS

Named parameters are just like normal parameters, but rather than using the specific order of the values provided to a function to determine which value belongs to which parameter (the first value is the first parameter, the second value is the second parameter, and so on), we explicitly name the values so they can appear in any order.

Sometimes functions have a lot of parameters, and we may not always need to provide a value for every one. With named parameters, we can provide values for only the parameters we need to give values.

For example, suppose we have a `person` function that takes two parameters, `width` and `height` :

```
>>> def person(width, height):
        print(f'I am {width} feet wide, {height} feet high')
```

Normally, we might call this function like so:

```
>>> person(4, 3)
I am 4 feet wide, 3 feet high
```

Using named parameters, we could call this function and specify the parameter name with each value:

```
>>> person(height=3, width=4)
I am 4 feet wide, 3 feet high
```

Named parameters will become particularly useful as we do more with the `tkinter` module.

CREATING A CANVAS FOR DRAWING

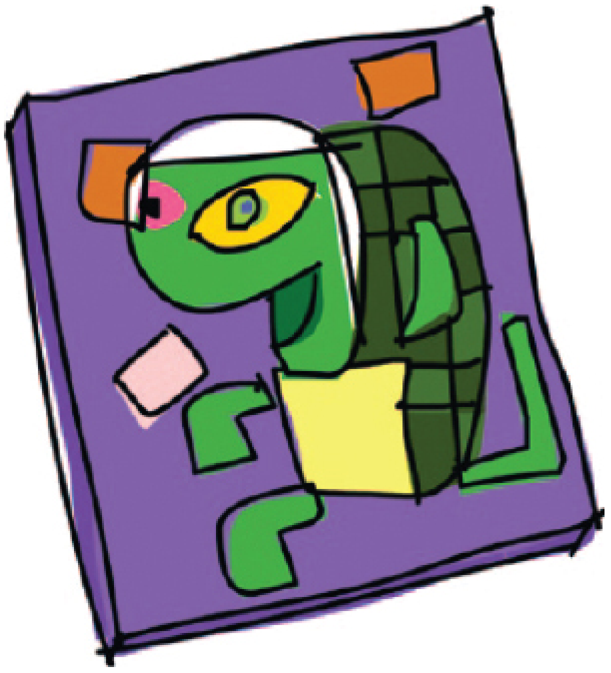
Buttons are nice tools, but they're not particularly useful when we want to draw things on the screen. When it's time to draw, we need a different component: a `canvas` object, which is an object of the `Canvas` class (provided by the `tkinter` module).

When creating a canvas, we pass the width and height (in pixels) of the canvas to Python. Otherwise, the code is similar to the button code. For example:

```
>>> from tkinter import *
>>> tk = Tk()
>>> canvas = Canvas(tk, width=500, height=500)
>>> canvas.pack()
```

As with the button example, a window will appear when you enter `tk = Tk()` . On the last line, we pack the canvas with `canvas.pack()` , which applies the changes to the size of the canvas (a width of 500 pixels and a height of 500 pixels, as specified in the third line of code).

Also like the button example, the `pack` function tells the canvas to display itself in the correct position within the window. If `pack` isn't called, nothing will display properly.



DRAWING LINES

To draw a line on the canvas, we use pixel coordinates. *Coordinates* determine the positions of pixels on a surface. On a `tkinter` canvas, coordinates describe how far across the canvas (from left to right) and how far down the canvas (top to bottom) to place the pixel.

For example, because our canvas is 500 pixels wide by 500 pixels high, the coordinates of the bottom-right corner of the screen are (500, 500). To draw the line shown in Figure 10-3 , we would use the starting coordinates (0, 0) and ending coordinates (500, 500).

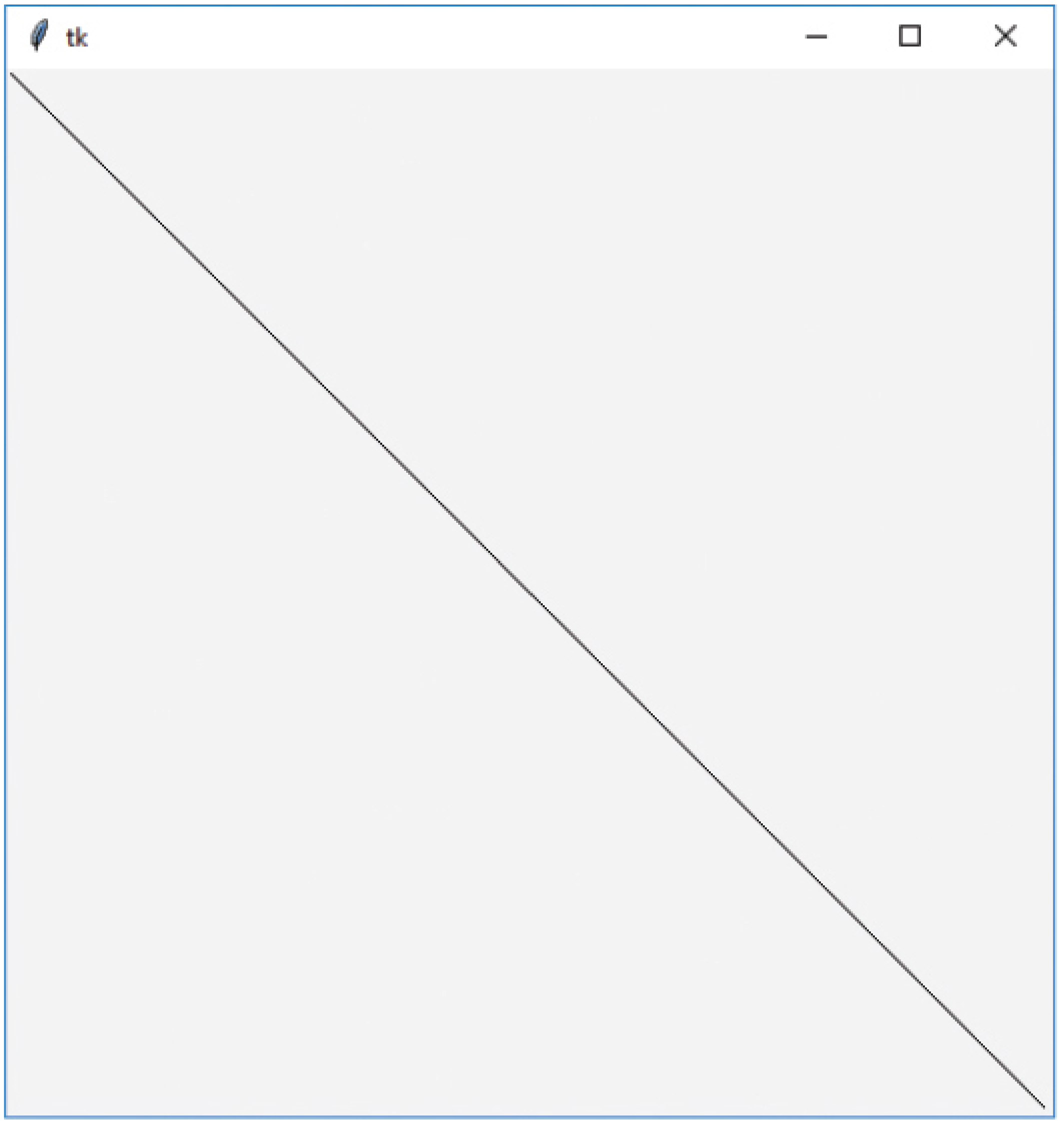


Figure 10-3: Drawing a diagonal line with `tkinter`

We specify the coordinates by using the `create_line` function, as shown here:

```
>>> from tkinter import *
>>> tk = Tk()
>>> canvas = Canvas(tk, width=500, height=500)
>>> canvas.pack()
>>> canvas.create_line(0, 0, 500, 500)
1
```

The `create_line` function returns 1, which is an identifier; we'll learn more about that later. If we had done the same thing with the `turtle` module, we would've needed the following this code:

```
>>> import turtle
>>> turtle.setup(width=500, height=500)
>>> t = turtle.Turtle()
>>> t.up()
>>> t.goto(-250, 250)
>>> t.down()
>>> t.goto(500, -500)
```

In this code, the canvas is 500 pixels wide and 500 high, so the turtle appears at position `250, 250` (in the middle of the canvas). If we use the function `t.goto(-250, 250)` , we're moving left 250 and up 250 pixels to the top left of the screen. When we call `t.goto(500, -500)` , we're then moving right 500 pixels and down 500 pixels to the bottom-right corner.

So we can see that the `tkinter` code is already an improvement. It's slightly shorter and simpler. Now let's look at some of the functions available on the `canvas` object that we can use to create more interesting drawings.

DRAWING BOXES

With the `turtle` module, we drew a box by moving forward, turning, moving forward, turning again, and so on. Eventually, we were able to draw a rectangular or square box by changing how far we moved forward.

The `tkinter` module makes it a lot easier to draw a square or rectangle. All you need to know are the coordinates for the corners. Try the following example (you can close the other windows now):

```
>>> from tkinter import *
>>> tk = Tk()
>>> canvas = Canvas(tk, width=400, height=400)
>>> canvas.pack()
>>> canvas.create_rectangle(10, 10, 50, 50)
```

In this code, we use `tkinter` to create a canvas that is 400 pixels wide by 400 pixels high, and we then draw a square in the top-left corner of the window, like Figure 10-4 .

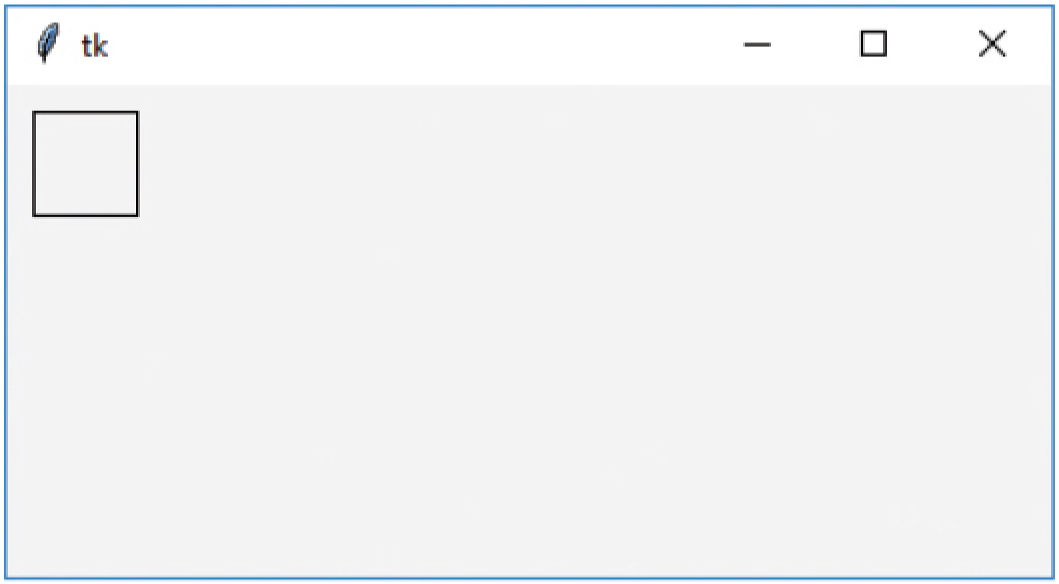


Figure 10-4: Drawing a box

The parameters we pass to `canvas.create_rectangle` in the last line of the code are the coordinates for the top-left and bottom-right corners of the square. We provide these coordinates as the distance from the left-hand side of the canvas and the distance from the top of the canvas. In this case, the first two coordinates (the top-left corner) are 10 pixels across from the left and 10 pixels down from the top—those are the first numbers: `10, 10` . The bottom-right corner of the square is 50 pixels across from the left and 50 pixels down from the top—the second numbers: `50, 50` .

We'll refer to these two sets of coordinates as *x1* , *y1* and *x2* , *y2* . To draw a rectangle, we can increase the distance of the second corner from the side of the canvas (increasing the value of the *x2* parameter), like this:

```
>>> from tkinter import *
>>> tk = Tk()
>>> canvas = Canvas(tk, width=400, height=400)
>>> canvas.pack()
>>> canvas.create_rectangle(10, 10, 300, 50)
```

In this example, the top-left coordinates of the rectangle (its position on the screen) are (`10, 10`), and the bottom-right coordinates are (`300, 50`). The result is a rectangle that is the same height as our original square (40 pixels) but much wider.

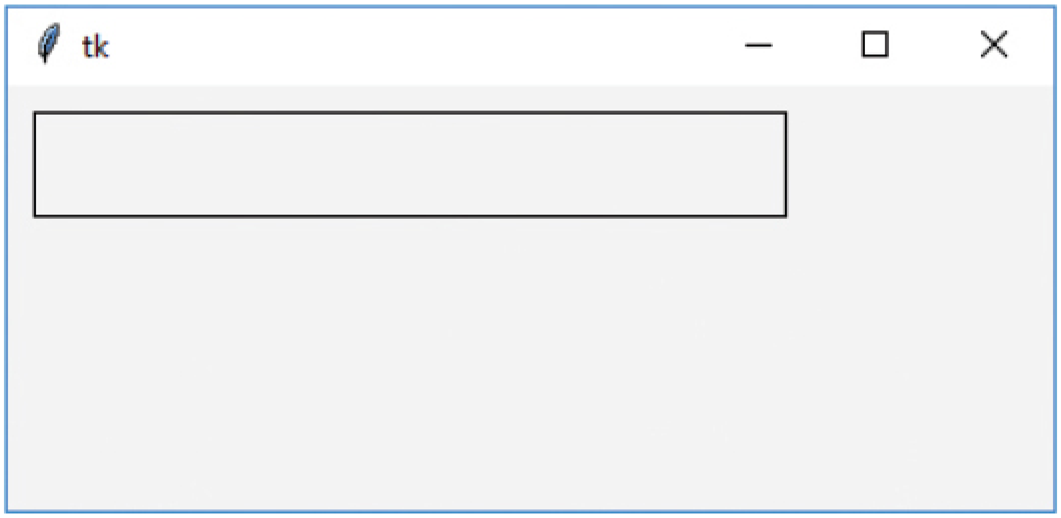


Figure 10-5: A wide rectangle

We can also draw a rectangle by increasing the distance of the second corner from the top of the canvas (increasing the value of the *y2* parameter), like this:

```
>>> from tkinter import *
>>> tk = Tk()
>>> canvas = Canvas(tk, width=400, height=400)
>>> canvas.pack()
>>> canvas.create_rectangle(10, 10, 50, 300)
```

In this call to the `create_rectangle` function, we're telling `tkinter` to:

- Go 10 pixels across the canvas (from the top left).
- Go 10 pixels down the canvas. This is the starting corner of the rectangle.
- Draw the rectangle across to 50 pixels.
- Draw down to 300 pixels.

The end result should look something like Figure 10-6 .

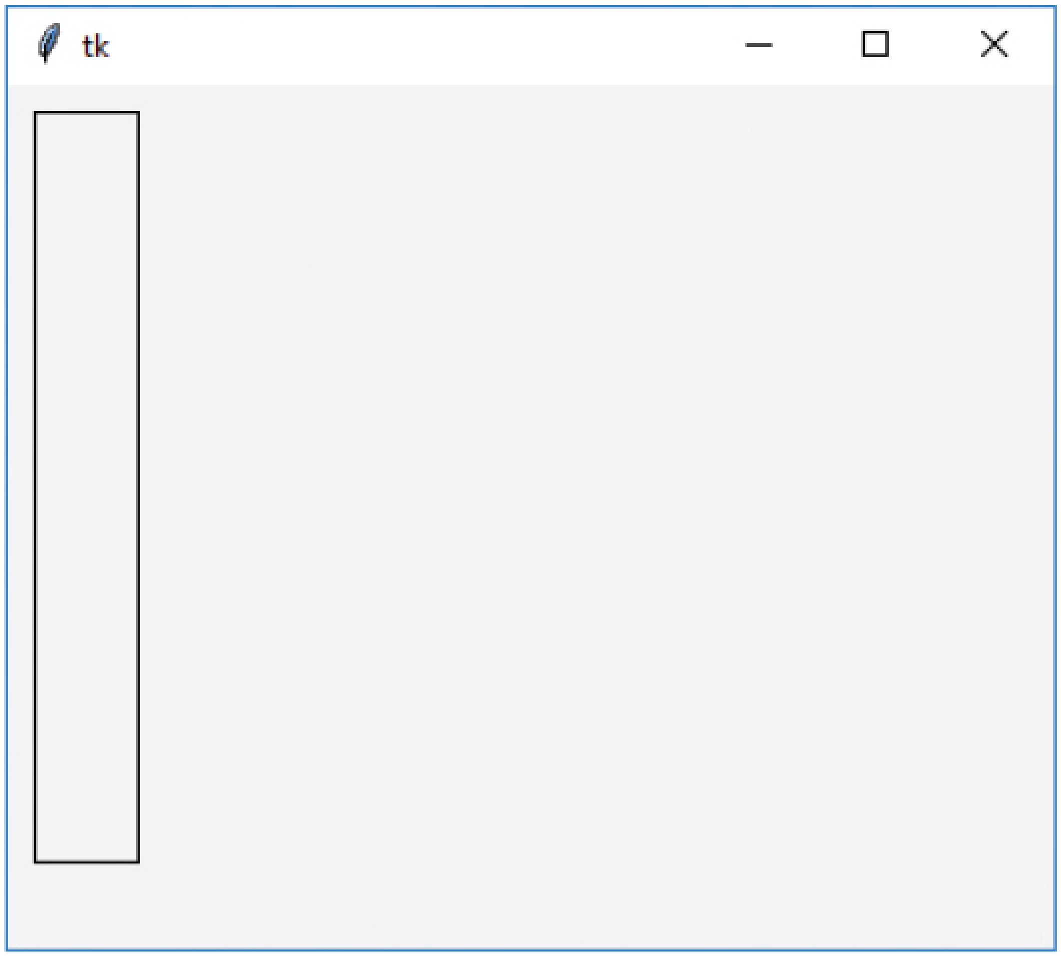


Figure 10-6: A tall rectangle

DRAWING A LOT OF RECTANGLES

Let's try filling the canvas with different-sized rectangles by importing the `random` module and then creating a function that uses a random number for the coordinates at the top-left and bottom-right corners of the rectangle.

We'll use the `randrange` function provided by the `random` module. When we give this function a number, it returns a random integer between 0 and the number we give it. For example, calling `randrange(10)` returns a number between 0 and 9, `randrange(100)` returns a number between 0 and 99, and so on.

To use `randrange` in a function, create a new window by selecting **File** ▸ **New File** , and enter the following code:

```
from tkinter import *
import random
tk = Tk()
canvas = Canvas(tk, width=400, height=400)
canvas.pack()

def random_rectangle(width, height):
    x1 = random.randrange(width)
    y1 = random.randrange(height)
    x2 = x1 + random.randrange(width)
    y2 = y1 + random.randrange(height)
    canvas.create_rectangle(x1, y1, x2, y2)
```

We first define our (`random_rectangle`) function as taking two parameters: `width` and `height` . Next, we create variables for the top-left corner of the rectangle by using the `randrange` function, passing the width and the height as parameters with `x1 = random .randrange(width)` and `y1 = random.randrange(height)` , respectively. With the second line of this function, we're saying, "Create a variable called `x1` and set its value to a random number between 0 and the value in the parameter `width` ."

The next two lines create variables for the bottom-right corner of the rectangle, taking into account the top-left coordinates (either `x1` or `y1`) and adding a random number to those values. The third line of the function is effectively saying, "Create the variable `x2` by adding a random number to the value we already calculated for `x1` ."

Finally, with `canvas.create_rectangle` , we use the variables `x1` , `y1` , `x2` , and `y2` to draw the rectangle on the canvas.

To try our `random_rectangle` function, we'll pass it the width and height of the canvas. Add the following code below the function you've just entered:

```
random_rectangle(400, 400)
```

Save the code you've entered (select **File** ▸ **Save** and enter a filename such as *randomrect.py*), and then select **Run** ▸ **Run Module** .

NOTE

Our `random_rectangle` *function can draw a rectangle off the side or bottom of the canvas. That's because the top-left corner of the rectangle can be anywhere on the canvas (even in the bottom right-hand corner), and it doesn't cause any errors to draw past the width or height of the canvas.*

Once you've seen the function working, fill the screen with rectangles by creating a loop to call `random_rectangle` a number of times. Let's try a `for` loop of 100 random rectangles. Add the following code, save your work, and try running it again:

```
for x in range(0, 100):
    random_rectangle(400, 400)
```

This code produces a bit of a mess, but it's kind of modern art (Figure 10-7).

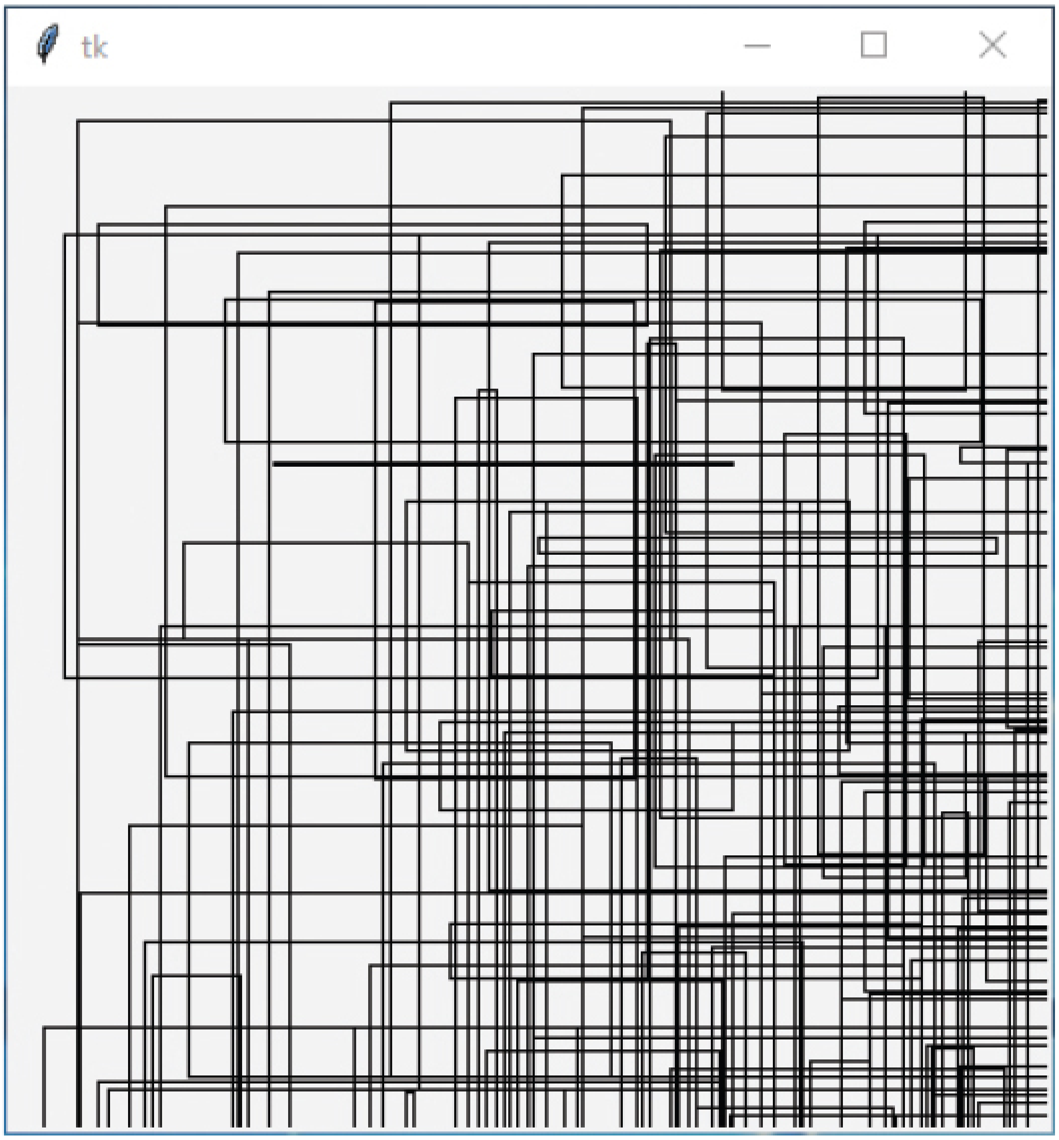


Figure 10-7: Modern art with tkinter

SETTING THE COLOR

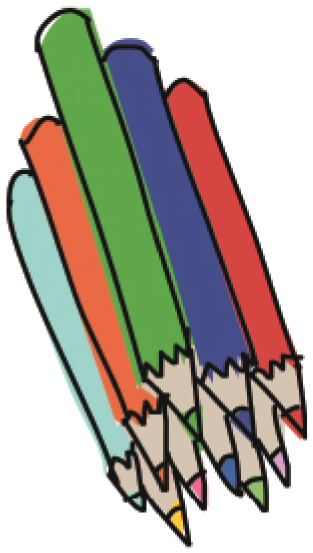
Let's add interest to our graphics with color. We'll change the `random_rectangle` function to pass in a color for the rectangle as an additional parameter (`fill_color`). Enter this code in a new window, and when you save, call the file *colorrect.py* :

```
from tkinter import *
import random
```

```
tk = Tk()
canvas = Canvas(tk, width=400, height=400)
canvas.pack()

def random_rectangle(width, height, fill_color):
    x1 = random.randrange(width)
    y1 = random.randrange(height)
    x2 = random.randrange(x1 + random.randrange(width))
    y2 = random.randrange(y1 + random.randrange(height))
    canvas.create_rectangle(x1, y1, x2, y2, fill=fill_color)
```

The `create_rectangle` function now takes a parameter, `fill_color` , which specifies the color to use when drawing the rectangle.

We can pass named colors into the function like this to create a bunch of uniquely colored rectangles. If you try this example, consider copying and pasting, after you enter the first line, to save on typing. To do so, select the text to copy, press CTRL-C to copy it, click a blank line, and press CTRL-V to paste. Add this code to *colorrect.py* , just below the function:



```
random_rectangle(400, 400, 'green')
random_rectangle(400, 400, 'red')
random_rectangle(400, 400, 'blue')
random_rectangle(400, 400, 'orange')
random_rectangle(400, 400, 'yellow')
random_rectangle(400, 400, 'pink')
random_rectangle(400, 400, 'purple')
random_rectangle(400, 400, 'violet')
random_rectangle(400, 400, 'magenta')
random_rectangle(400, 400, 'cyan')
```

Many of these named colors will display the color you expect to see, but others may produce an error message (depending on whether you're using Windows, macOS, or Linux). But what about a custom color that isn't exactly the same as a named color? Recall in Chapter 9 that we set the color of the turtle's pen by using percentages of the colors red, green, and blue. Setting the amount of each primary color (red, green, and blue) to use in a color combination with `tkinter` is slightly more complicated, but we'll work through it.

When working with the `turtle` module, we created gold using 90 percent red, 75 percent green, and no blue. In `tkinter` , we can create the same gold color by using this line:

```
random_rectangle(400, 400, '#e5d800')
```

The hash mark (`#`) before the value `ffd800` tells Python we're providing a *hexadecimal* number. Hexadecimal is a way of representing numbers that is common in computer programming. It uses a base of 16 (0 through 9, then A through F) rather than decimal, which has a base of 10 (0 through 9). If you haven't learned about bases in mathematics, just know that you can convert a normal decimal number to hexadecimal using a *format placeholder* in a string: `{:x}` (see "Embedding Values in Strings" on page 29). For example, to convert the decimal number 15 to hexadecimal, you could do this:

```
>>> print(f'{15:x}')
f
```

This is an f-string with a special format modifier (that's `:x`) that tells Python to convert the number to hexadecimal.

To make sure our number has at least two digits, we can change the format placeholder slightly, to this:

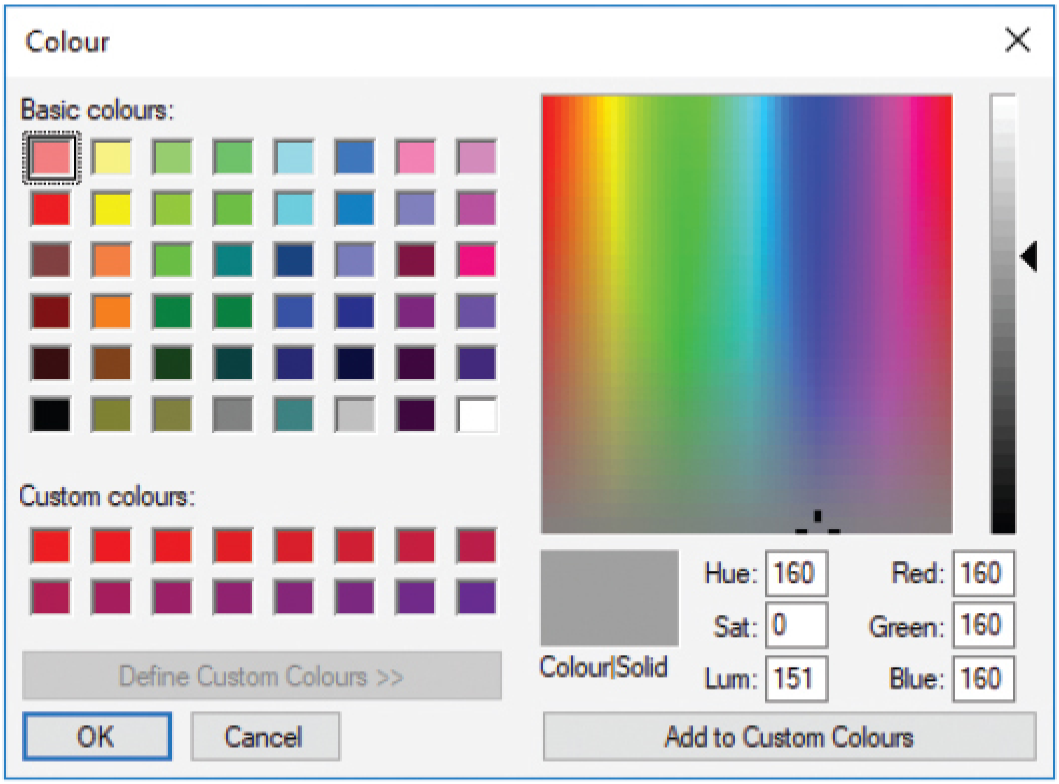
```
>>> print(f'{15:02x}')
0f
```

This time we have a slightly different format modifier (`02x`) that says we want hexadecimal conversion, but with two digits (using 0 for any missing digit).

The `tkinter` module provides an easy way to get hexadecimal color values. Try running the following code in IDLE:

```
from tkinter import *
from tkinter import colorchooser
tk = Tk()
tk.update()
print(colorchooser.askcolor())
```

This code displays a color chooser, shown in Figure 10-8 . Note that you have to explicitly import the `colorchooser` module because it's not automatically available in Python when you use `from tkinter import *` .



Hue: 160
Sat: 0
Lum: 151
Red: 160
Green: 160
Blue: 160

Figure 10-8: The tkinter color chooser (which may look different on your OS)

When you select a color and click **OK** , a tuple will be displayed. This tuple contains another tuple with three numbers and a string:

```
>>> print(colorchooser.askcolor())
((157, 163, 164), '#9da3a4')
```

The three numbers represent the amounts of red, green, and blue. In `tkinter` , the amount of each primary color to use in a color combination is represented by a number between 0 and 255 (which is different from using a percentage for each primary color with the `turtle` module). The string in the tuple contains the hexadecimal version of those three numbers.

You can either copy and paste the string value to use it, or store the tuple as a variable and use the index position of the hexadecimal value.

Let's use the `random_rectangle` function to see how this works, by replacing all the `random_rectangle` calls at the bottom of *colorrect.py* with the following code:

```
from tkinter import colorchooser
c = colorchooser.askcolor()
random_rectangle(400, 400, c[1])
```

You can see the result in Figure 10-9 .

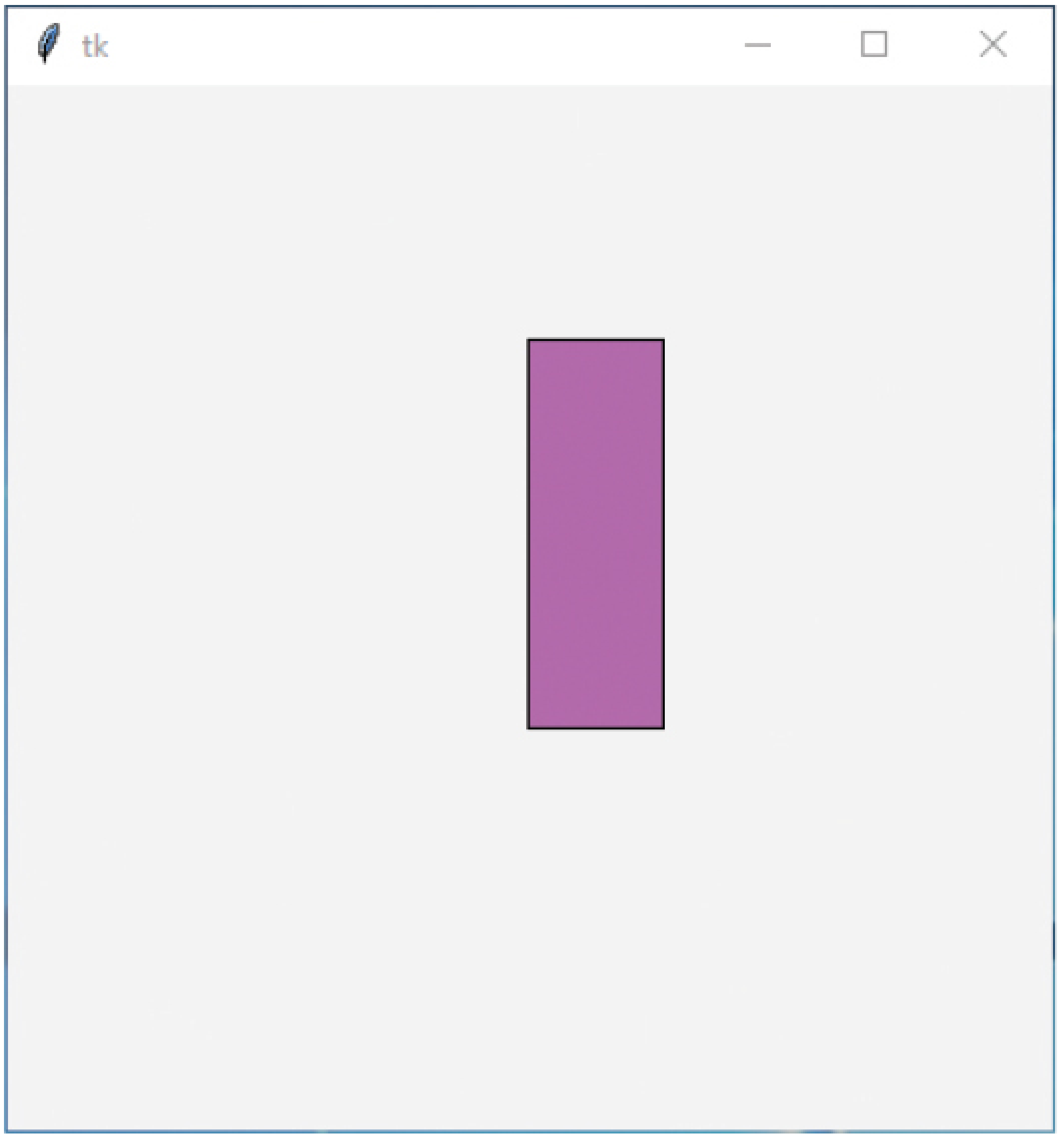


Figure 10-9: Drawing a purple rectangle

DRAWING ARCS

An *arc* is a segment of the circumference of a circle or curve. To draw an arc with `tkinter` , you need to draw it inside a rectangle by using the `create_arc` function with code like this:

```
canvas.create_arc(10, 10, 200, 100, extent=180, style=ARC)
```

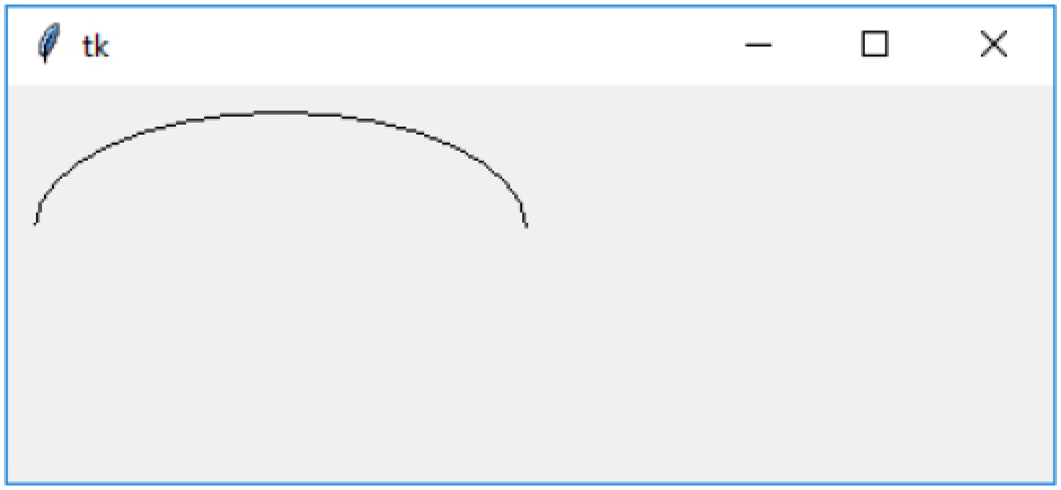


Figure 10-10: Drawing an arc

NOTE

If you've closed all the `tkinter` *windows or restarted IDLE, make sure to reimport* `tkinter` *and then re-create the canvas with this code:*

```
>>> from tkinter import *
>>> tk = Tk()
>>> canvas = Canvas(tk, width=400, height=400)
>>> canvas.pack()
>>> canvas.create_arc(10, 10, 200, 100, extent=180, style=ARC)
```

This code places the top-left corner of the rectangle that will contain the arc at the coordinates (`10, 10`), which is 10 pixels across and 10 pixels down, and its bottom-right corner at coordinates (`200, 100`), or 200 pixels across and 100 pixels down. The next parameter, `extent` , is used to specify the degrees of the angle of the arc. Recall from Chapter 4 that degrees are a way of measuring the distance to travel around a circle. Figure 10-11 shows examples of two arcs, where we travel 90 degrees and 270 degrees around a circle.

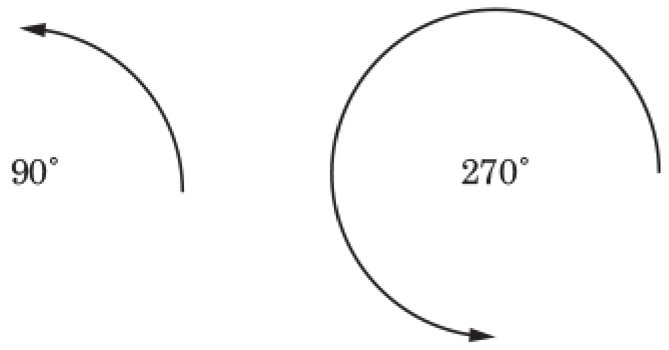


Figure 10-11: 90- and 270- degree arcs

The following code draws several different arcs down the page so you can see what happens when we use different degrees with the `create_arc` function:

```
>>> from tkinter import *
>>> tk = Tk()
>>> canvas = Canvas(tk, width=400, height=400)
>>> canvas.pack()
>>> canvas.create_arc(10, 10, 200, 80, extent=45, style=ARC)
>>> canvas.create_arc(10, 80, 200, 160, extent=90, style=ARC)
>>> canvas.create_arc(10, 160, 200, 240, extent=135, style=ARC)
>>> canvas.create_arc(10, 240, 200, 320, extent=180, style=ARC)
>>> canvas.create_arc(10, 320, 200, 400, extent=359, style=ARC)
```

The result is shown in Figure 10-12 .

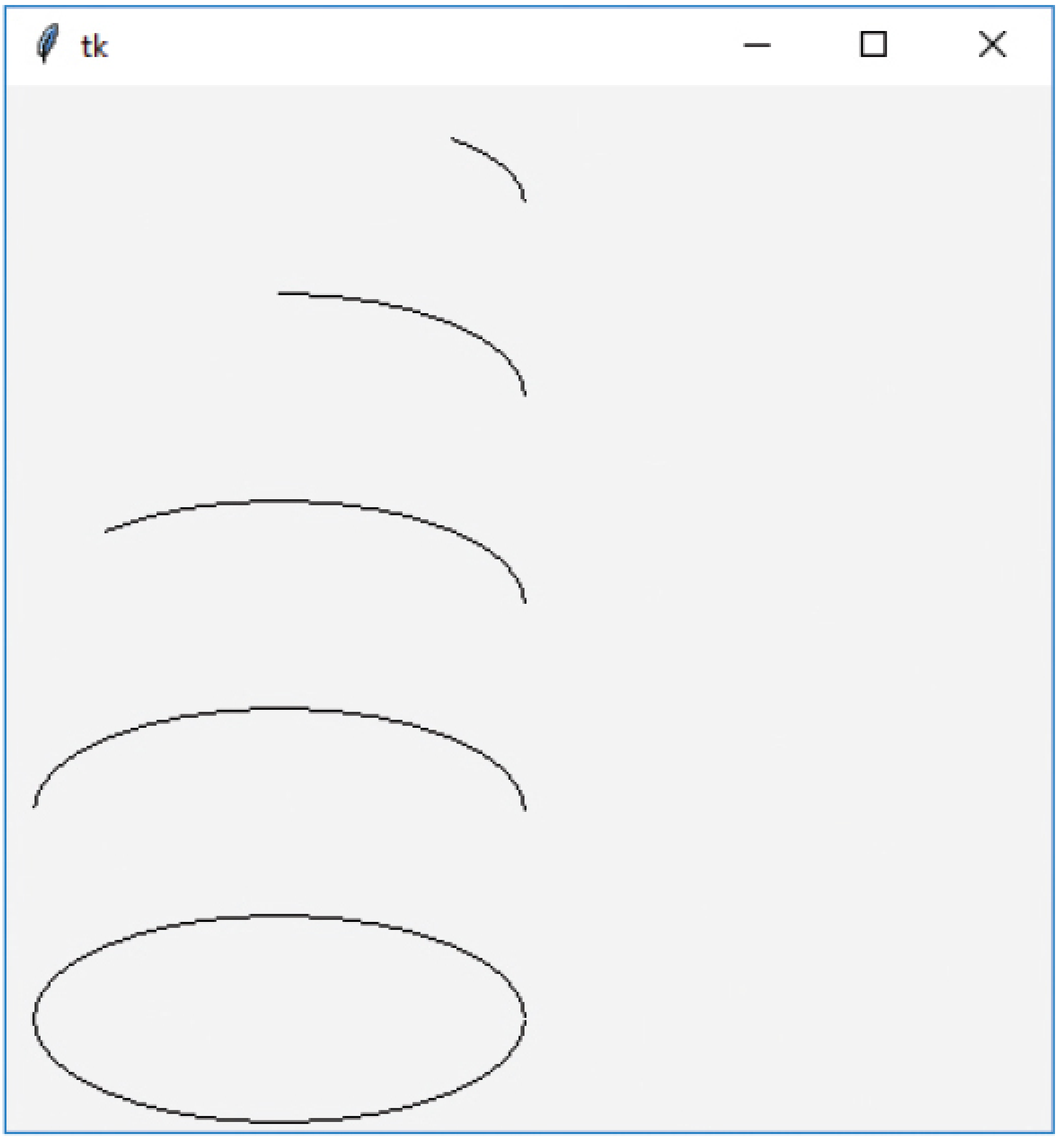


Figure 10-12: Multiple arcs

NOTE

We use 359 degrees in the final circle, rather than 360, because `tkinter` *considers 360 to be the same as 0 degrees, and would draw nothing.*

The `style` parameter is the type of arc you want to draw. There are two other types of arc: chord and pieslice. A *chord* is almost the same as the arc we have already drawn, except the two ends are joined together with a single straight line. A *pieslice* is exactly what it sounds like—as if you cut a segment out of a pizza or a pie.

DRAWING POLYGONS

A *polygon* is any shape with three or more sides. There are regularly shaped polygons—like triangles, squares, rectangles, pentagons, hexagons, and so on—as well as *irregular* polygons with uneven edges, many more sides, and odd shapes.

When drawing polygons with `tkinter` , you need to provide coordinates for each point of the polygon. We can draw a triangle with the following code:

```
>>> from tkinter import *
>>> tk = Tk()
>>> canvas = Canvas(tk, width=400, height=400)
>>> canvas.pack()
>>> canvas.create_polygon(10, 10, 100, 10, 100, 110, fill='',
    outline='black')
```

This example draws a triangle by starting with the *x* and *y* coordinates (10, 10), then moving across to (`100, 10`), and finishing at (`100, 110`). We set the fill color to nothing (an empty string), so the triangle won't be colored in, and the outline is set to `'black'` , so it will be drawn with a black line. It should look like Figure 10-13 .

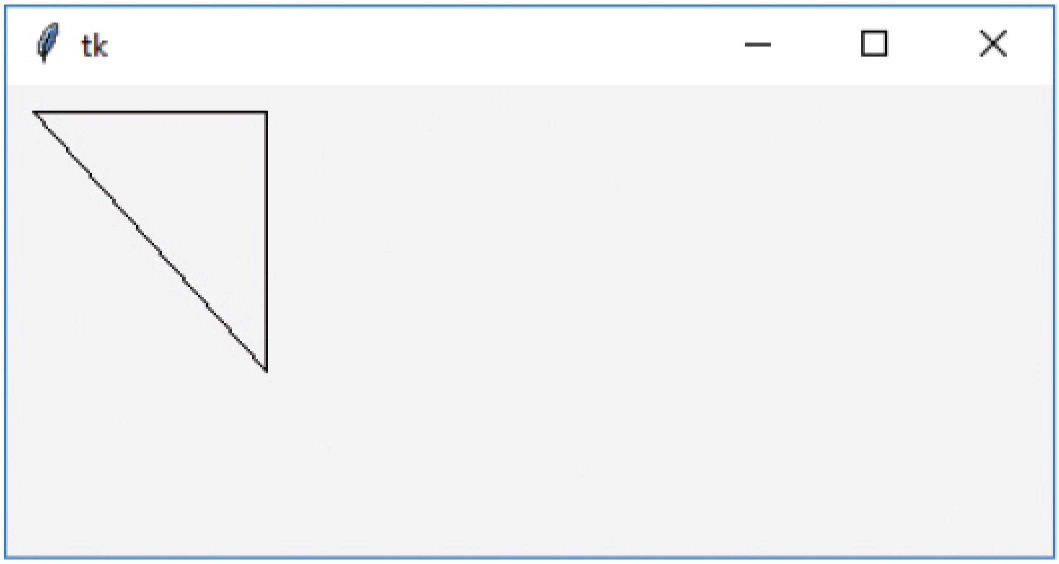


Figure 10-13: Drawing a triangle

We can add an irregular polygon by using this code:

```
canvas.create_polygon(200, 10, 240, 30, 120, 100, 140, 120, fill='',
outline='black')
```

This code begins with the coordinates (`200, 10`), moves to (`240, 30`), then to (`120, 100`), and finally to (`140, 120`). The `tkinter` module automatically joins the line back to the first coordinate. The result is shown in Figure 10-14 .

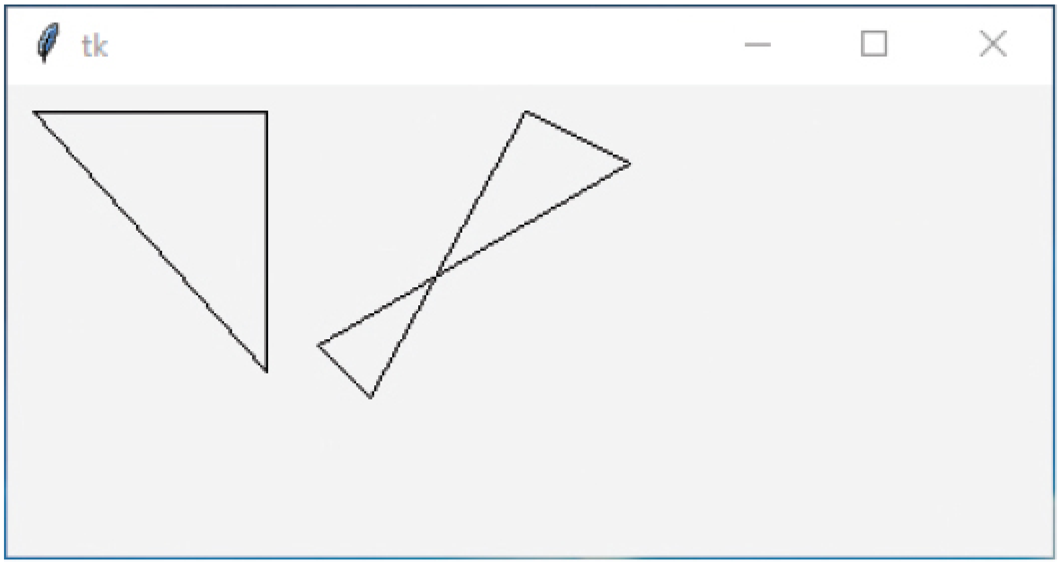


Figure 10-14: Irregular polygon

DISPLAYING TEXT

In addition to drawing shapes, you can also write on the canvas by using the `create_text` function. This function takes only two coordinates—the *x* and *y* positions of the text—along with a named parameter for the text to display. In the following code, we create our canvas as before and then display a sentence positioned at the coordinates (`150, 100`). Save this code as *text.py* :

```
from tkinter import *
tk = Tk()
canvas = Canvas(tk, width=400, height=400)
canvas.pack()
canvas.create_text(150, 100, text='There once was a man from Toulouse,')
```

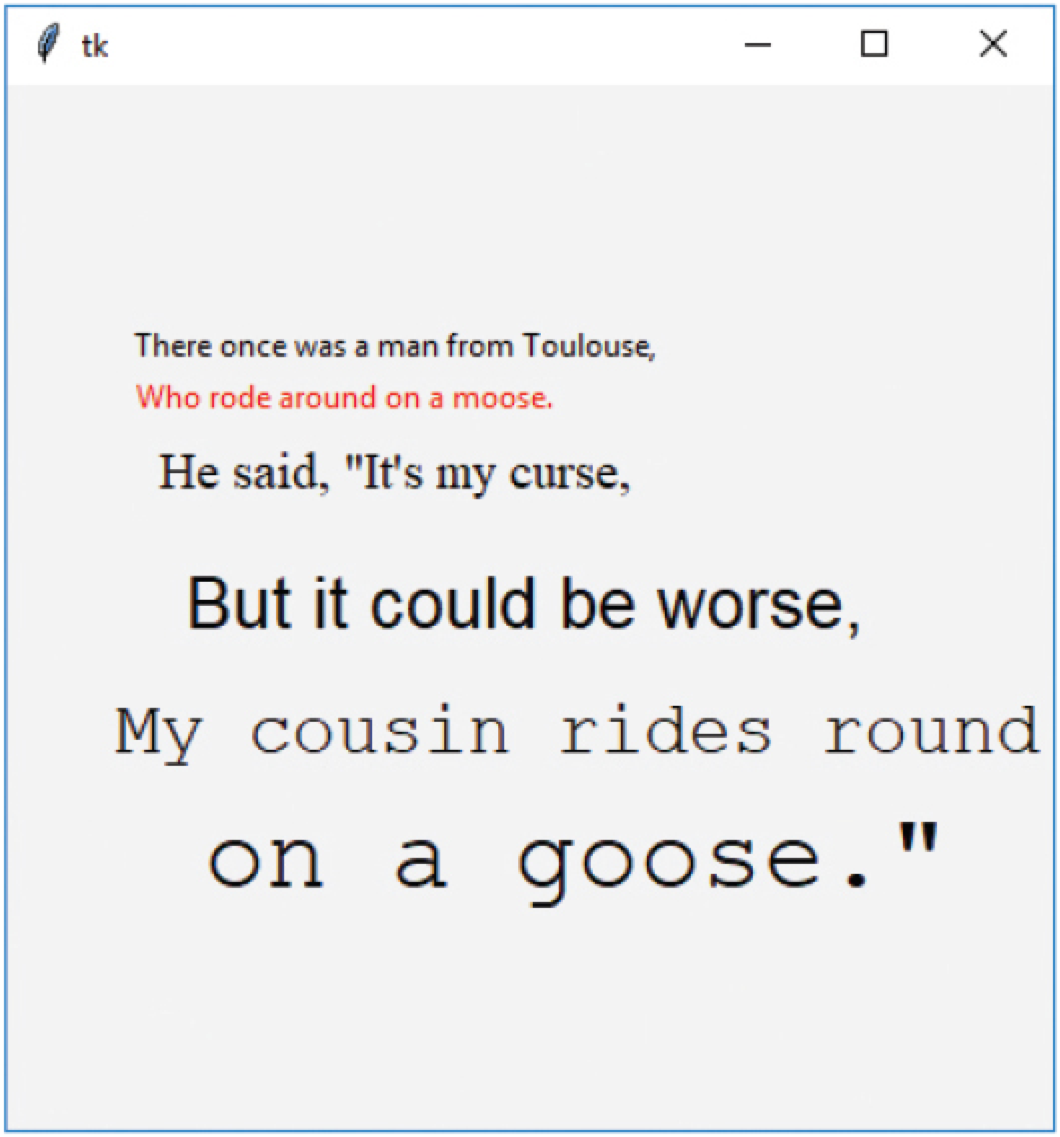
The `create_text` function can take other useful parameters, such as a text fill color. In the following code, we call the `create _text` function with coordinates (`130, 120`), the text we want to display, and a red fill color:

```
canvas.create_text(130, 120, text='Who rode around on a moose.', fill='red')
```

You can also specify the *font* , or the typeface used for the displayed text, as a tuple with the font name and the size of the text. For example, the tuple for the *Times* font of size 20 is (`'Times', 20`). In the following code, we display text using the *Times* font set at size 15, the *Helvetica* font at size 20, and the *Courier* font at sizes 22 and then 30:

```
canvas.create_text(150, 150, text='He said, "It\'s my curse,', font=('Times', 15))
canvas.create_text(200, 200, text='But it could be worse,', font=('Helvetica', 20))
canvas.create_text(220, 250, text='My cousin rides round', font=('Courier', 22))
canvas.create_text(220, 300, text='on a goose."', font=('Courier', 30))
```

Figure 10-15 shows the result of these functions using the three specified fonts at five different sizes.



```
There once was a man from Toulouse,
Who rode around on a moose.
He said, "It's my curse,
But it could be worse,
My cousin rides round
on a goose."
```

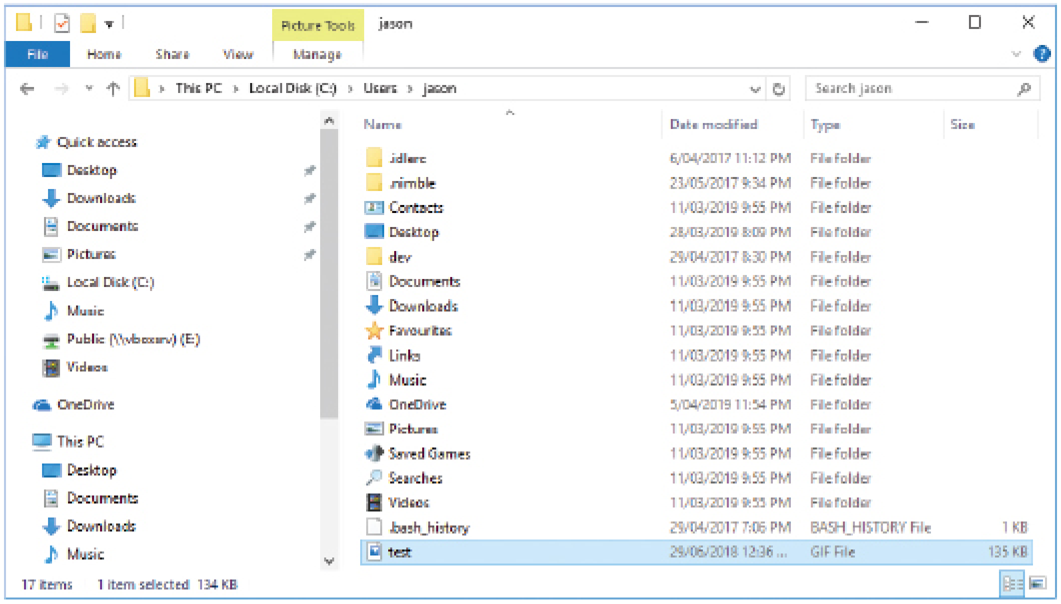
Figure 10-15: Drawing text with `tkinter`

DISPLAYING IMAGES

To display an image on a canvas by using `tkinter` , first load the image and then use the `create_image` function on the canvas object. Any image you load must be in a folder (or directory) that's accessible to Python.

The best place to put images is in your home folder. On Windows this is *c:* \ *Users* \ *<your username>* ; on macOS, */Users/<your username>* ;

and on Ubuntu or Raspberry Pi, */home/<your username>* . Figure 10-16 shows a home folder on Windows.



This PC › Local Disk (C:) › Users › jason

Name	Date modified	Type	Size
.idlerc	6/04/2017 11:12 PM	File folder	
.nimble	23/05/2017 9:34 PM	File folder	
Contacts	11/03/2019 9:55 PM	File folder	
Desktop	28/03/2019 8:09 PM	File folder	
dev	29/04/2017 8:30 PM	File folder	
Documents	11/03/2019 9:55 PM	File folder	
Downloads	11/03/2019 9:55 PM	File folder	
Favourites	11/03/2019 9:55 PM	File folder	
Links	11/03/2019 9:55 PM	File folder	
Music	11/03/2019 9:55 PM	File folder	
OneDrive	5/04/2019 11:54 PM	File folder	
Pictures	11/03/2019 9:55 PM	File folder	
Saved Games	11/03/2019 9:55 PM	File folder	
Searches	11/03/2019 9:55 PM	File folder	
Videos	11/03/2019 9:55 PM	File folder	
.bash_history	29/04/2017 7:06 PM	BASH_HISTORY File	1 KB
test	29/06/2018 12:36 ...	GIF File	135 KB

17 items 1 item selected 134 KB

Figure 10-16: Home folder on Windows

NOTE

With `tkinter` *, you can load only GIF images—image files with the extension* .gif*. You can display other types of images, such as PNG* (.png) *and JPG* (.jpg)*, but you'll need to use a different module such as Pillow, the Python Imaging Library (found at* https://python-pillow.org *). If you don't have a GIF to use, try opening a photo and then saving it as a GIF. On Windows you can do this quite easily with the Paint app—but there are plenty of other ways to convert images to the GIF format.*

We can display an image called *test.gif* as follows.

```
from tkinter import *
tk = Tk()
canvas = Canvas(tk, width=400, height=400)
canvas.pack()
my_image = PhotoImage(file='c:\\Users\\jason\\test.gif')
canvas.create_image(0, 0, anchor=NW, image=my_image)
```

In the first four lines, we set up the canvas as with the previous examples. In the fifth line, the image is loaded into the `my_image` variable. We create `PhotoImage` with the filename *c:\\ Users\\jason\\test.gif* . We need to use two backslashes (\\ ) in a Windows filename, because backslash is a special character in a Python string (used for something called an escape character—for example, *\t* is the escape character representing a tab, *\n* is the escape character representing a newline, which we used back in Chapter 7), and two backslashes are simply a way of saying, “I don’t want to use an escape character here—I want a single backslash.”

If you saved your image to the desktop, you should create the `PhotoImage` with that folder, like this:

```
my_image = PhotoImage(file='C:\\Users\\JoeSmith\\Desktop\\test.gif')
```

Once the image has been loaded into the variable, `canvas.create_image(0, 0, anchor=NW, image=my_image)` displays it using the `create_image` function. The coordinates (`0, 0`) are where the image will be displayed, and `anchor=NW` (with NW standing for *northwest*) tells the function to use the top-left edge of the image as the starting point when drawing; otherwise, it will use the center of the image as the starting point by default. The final named parameter, `image` , points at the variable for the loaded image. Your screen should look similar to Figure 10-17 .

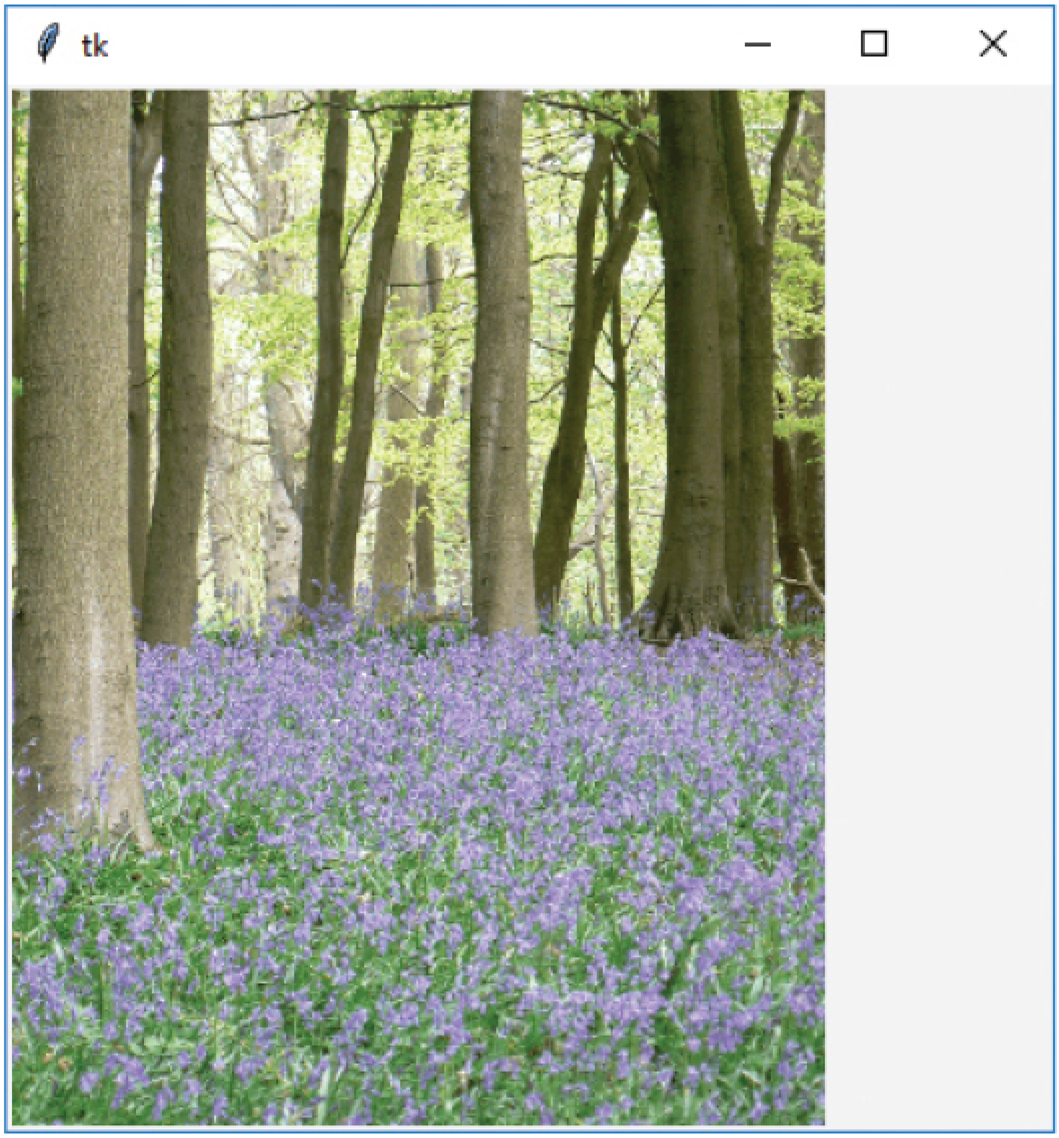


Figure 10-17: Displaying an image

CREATING BASIC ANIMATION

We've covered how to create static pictures that don't move. Now, we'll turn our attention to creating animation.

Animation is not necessarily a specialty of the `tkinter` module, but it can handle the basics. For example, we can create a filled triangle and then make it move across the screen by using this code:

```
>>> import time
>>> from tkinter import *
>>> tk = Tk()
>>> canvas = Canvas(tk, width=400, height=200)
>>> canvas.pack()
>>> canvas.create_polygon(10, 10, 10, 60, 50, 35)
>>> for x in range(1, 61):
        canvas.move(1, 5, 0)
        tk.update()
        time.sleep(0.05)
```

When you run this code, the triangle will start moving across the screen to the end of its path, as in Figure 10-18 .

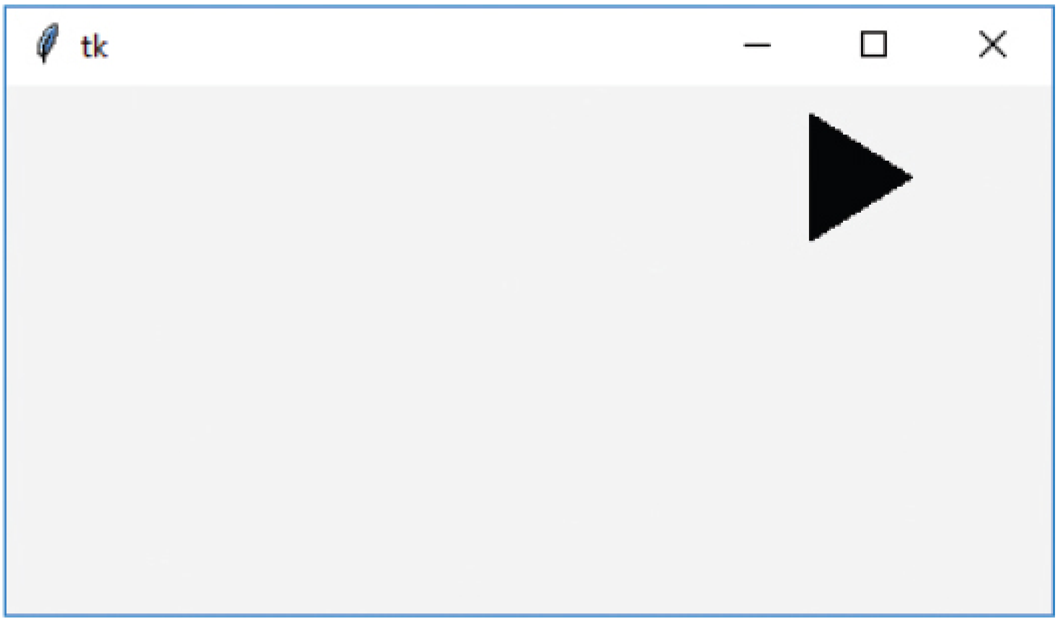


Figure 10-18: Moving triangle

As before, we've used the first three lines after importing `tkinter` to do the basic setup to display a canvas. We create the triangle with the call to the `canvas.create_polygon(10, 10, 10, 60, 50, 35)` function.

NOTE

When you enter this line, a number will be printed to the screen. This is an identifier for the polygon. We can use it to refer to the shape later, as described in the following example.

Next, we create a simple `for` loop to count from 1 to 61, beginning with `for x in range(1, 61)` .

The block of code inside the loop moves the triangle across the screen. The `canvas.move` function will move any drawn object by adding values to its `x` and `y` coordinates. For example, with `canvas.move(1, 5, 0)` , we move the object with ID 1 (the identifier for the triangle—see the preceding Note) 5 pixels across and 0 pixels down. To move it back again, we could use the function call `canvas.move(1, -5, 0)` .

The `tk.update()` function forces `tkinter` to update the screen (redraw it). If we didn't use `update` , `tkinter` would wait until the loop finished before moving the triangle, which means you would see it jump to the last position, rather than move smoothly across the canvas. The final line of the loop, `time.sleep(0.05)` , tells Python to sleep for one-twentieth of a second (0.05 seconds) before continuing.

To make the triangle move diagonally down the screen, we can modify this code by calling `move(1, 5, 5)` . Close the canvas and create a new file (**File** ▸ **New File**) for the following code:

```
import time
from tkinter import *
tk = Tk()
canvas = Canvas(tk, width=400, height=400)
canvas.pack()
canvas.create_polygon(10, 10, 10, 60, 50, 35)
for x in range(0, 60):
    canvas.move(1, 5, 5)
    tk.update()
    time.sleep(0.05)
```

This code differs from the original in two ways:

- We made the height of the canvas 400, rather than 200, with `canvas = Canvas(tk, width=400, height=400)` .
- We added 5 to the triangle's *x* and *y* coordinates with `canvas.move(1, 5, 5)` .

Figure 10-19 shows the triangle's position at the end of the loop, after you save your code and run it.

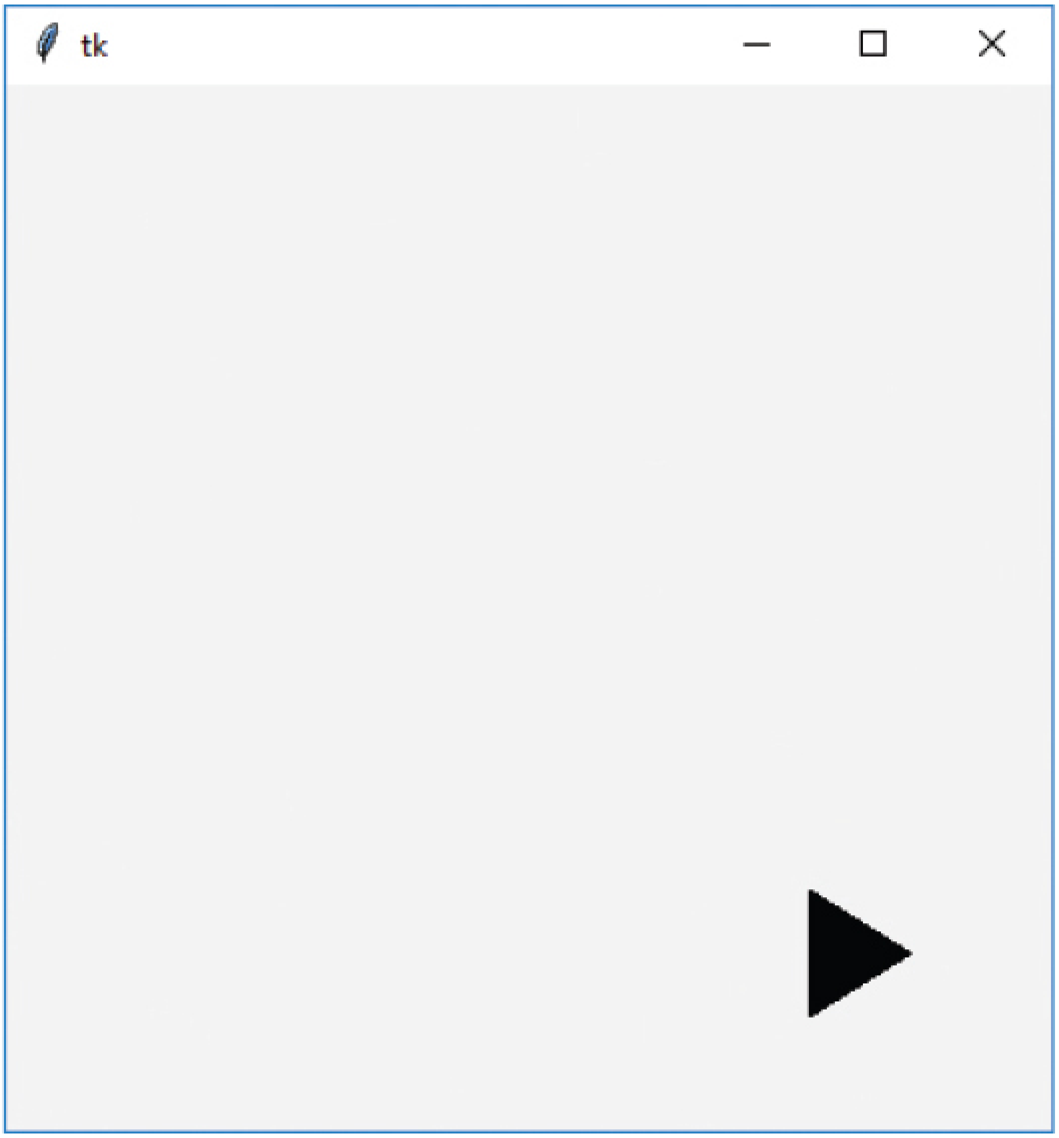


Figure 10-19: The triangle moved to the bottom of the screen.

To move the triangle diagonally back up the screen to its starting position, use (-5, -5). Add this code to the bottom of the file:

```
>>> for x in range(0, 60):
        canvas.move(1, -5, -5)
        tk.update()
        time.sleep(0.05)
```

After you run this code, the triangle will move back to where it started.

MAKING AN OBJECT REACT TO SOMETHING

We can make the triangle react when someone presses a key by using *event bindings* . *Events* are things that occur while a program is running, such as someone moving the mouse, pressing a key, or closing a window. You can tell tkinter to watch for these events and then do something in response.

To begin *handling* events (making Python do something when an event occurs), we first create a function. The binding part comes when we tell tkinter that a particular function is bound (or associated) to a specific event. In other words, it will be automatically called by tkinter to handle that event.

For example, to make the triangle move when we press ENTER, we can define this function:

```
def movetriangle(event):
    canvas.move(1, 5, 0)
```

The function takes a single parameter (event), which tkinter uses to send information to the function about the event. We tell tkinter that this function should be used for a particular event by using the bind_all function on the canvas. The full code now looks like this—let's type it into a new file in IDLE and save it as *movingtriangle.py* before we run it:

```
from tkinter import *
tk = Tk()
canvas = Canvas(tk, width=400, height=400)
canvas.pack()
canvas.create_polygon(10, 10, 10, 60, 50, 35)
def movetriangle(event):
    canvas.move(1, 5, 0)
canvas.bind_all('<KeyPress-Return>', movetriangle)
```

The first parameter in this function describes the event that we want tkinter to watch for. In this case, it's called <KeyPress-Return> , which is a press of the ENTER or RETURN key. We tell tkinter that the movetriangle function should be called whenever this KeyPress event occurs.

Run this code, click the canvas with your mouse, and then try pressing ENTER on your keyboard.

Let's try changing the direction of the triangle depending on different key presses, such as the arrow keys. We first need to change the `movetriangle` function to the following:

```
def movetriangle(event):
    if event.keysym == 'Up':
        canvas.move(1, 0, -3)
    elif event.keysym == 'Down':
        canvas.move(1, 0, 3)
    elif event.keysym == 'Left':
        canvas.move(1, -3, 0)
    else:
        canvas.move(1, 3, 0)
```

The event object passed to `movetriangle` contains several variables. One of these variables, `keysym` (for *key symbol*), is a string that holds the value of the actual key pressed. The line `if event .keysym == 'Up'` says that if the `keysym` variable contains the string `'Up'` , we should call `canvas.move` with the parameters (1, 0, –3), as we do in the following line. If `keysym` contains `'Down'` , as in `elif event.keysym == 'Down'` , we call it with the parameters (1, 0, 3), and so on.

Remember: The first parameter is the identifying number for the shape drawn on the canvas, the second is the value to add to the *x* (horizontal) coordinate, and the third is the value to add to the *y* (vertical) coordinate.

We then tell `tkinter` that the `movetriangle` function will be used to handle events from four different keys (up, down, left, and right). The following shows how the *movingtriangle.py* code should look now:

```
from tkinter import *
tk = Tk()
canvas = Canvas(tk, width=400, height=400)
canvas.pack()
canvas.create_polygon(10, 10, 10, 60, 50, 35)
def movetriangle(event):
 ❶ if event.keysym == 'Up':
     ❷ canvas.move(1, 0, -3)
 ❸ elif event.keysym == 'Down':
     ❹ canvas.move(1, 0, 3)
    elif event.keysym == 'Left':
        canvas.move(1, -3, 0)
 ❺ else:
     ❻ canvas.move(1, 3, 0)
canvas.bind_all('<KeyPress-Up>', movetriangle)
canvas.bind_all('<KeyPress-Down>', movetriangle)
canvas.bind_all('<KeyPress-Left>', movetriangle)
canvas.bind_all('<KeyPress-Right>', movetriangle)
```

On the first line of the `movetriangle` function, we check whether the `keysym` variable contains `'Up'` ❶ . If it does, we move the triangle upward using the `move` function with the parameters `1` , `0` , `{3` ❷ . The first parameter is the identifier of the triangle, the second is the amount to move to the right (we don't want to move horizontally, so the value is 0), and the third is the amount to move downward (–3 pixels).

We then check whether `keysym` contains `'Down'` ❸ ; if so, we move the triangle down (3 pixels) ❹ . The final check is whether the value is `'Left'` ; if so, we move the triangle left (–3 pixels). If none of the values are matched, the final `else` ❺ moves the triangle right ❻ .

Now the triangle should move in the direction of the pressed arrow key.

MORE WAYS TO USE THE IDENTIFIER

Whenever we use a `create_function` from the canvas, such as `create_polygon` or `create_rectangle` , an identifier is returned. This identifying number can be used with other canvas functions, as we did earlier with the `move` function:

```
>>> from tkinter import *
>>> tk = Tk()
>>> canvas = Canvas(tk, width=400, height=400)
>>> canvas.pack()
>>> canvas.create_polygon(10, 10, 10, 60, 50, 35)
1
>>> canvas.move(1, 5, 0)
```

The problem with this example is that `create_polygon` won't always return 1. For example, if you've created other shapes, it might return 2, 3, or even 100 (depending on the number of shapes you've created). If we change the code to store the value returned as a variable, and then use the variable (rather than just referring to the number 1), the code will work no matter what number is returned.

```
>>> mytriangle = canvas.create_polygon(10, 10, 10, 60, 50, 35)
>>> canvas.move(mytriangle, 10, 0)
```

The `move` function allows us to move objects around the screen by using their identifier. But other canvas functions can also change something we've drawn. For example, the `itemconfig` function can change some parameters of a shape, such as its fill and outline colors.

Say we create a red triangle:

```
>>> from tkinter import *
>>> tk = Tk()
>>> canvas = Canvas(tk, width=400, height=400)
>>> canvas.pack()
>>> mytriangle = canvas.create_polygon(10, 10, 10, 60, 50, 35,
    fill='red')
```

We can change the triangle to another color with `itemconfig` and use the identifier as the first parameter. The following code says, "Change the fill color of the object identified by the number in variable `mytriangle` to blue":

```
>>> canvas.itemconfig(mytriangle, fill='blue')
```

We could also give the triangle a different-colored outline, again using the identifier as the first parameter:

```
>>> canvas.itemconfig(mytriangle, outline='red')
```

Later, we'll learn how to make other changes to a drawing, like hiding it and making it visible again. You'll find the ability to change your drawings useful when we start writing games in the next chapter.

WHAT YOU LEARNED

In this chapter, you used the `tkinter` module to draw simple geometric shapes on a canvas, display images, and perform basic animation. You learned how event bindings can make drawings react to someone pressing a key, which will be helpful once we start programming a game. You learned how `tkinter` 's `create` functions return an identifying number, which can be used to modify shapes after they've been drawn, such as to move them around on the screen or change their color.

PROGRAMMING PUZZLES

Try the following to further explore the `tkinter` module and basic animation. Visit *http://python-for-kids.com* to download the solutions.

#1: FILL THE SCREEN WITH TRIANGLES

Create a program using `tkinter` to fill the screen with triangles. Then change the code to fill the screen with different-colored (filled) triangles instead.

#2: THE MOVING TRIANGLE

Modify the code for the moving triangle ("Creating Basic Animation" on page 159) to make it move across the screen to the right, then down, then back to the left, and then back to its starting position.

#3: THE MOVING PHOTO

Try displaying a photo of yourself on the canvas. Make sure it's a GIF image! Can you make it move across the screen?

#4: FILL THE SCREEN WITH PHOTOS

Take the photo you used in the previous puzzle, and shrink it down small.

On macOS, you can use Preview to resize an image (choose **Tools** ▸ **Adjust Size** , and enter a new width and height. Then, click **File** ▸ **Export** to save with a new filename).

On Windows, you can use Paint (click the **Resize** button, choose a horizontal and vertical size, then **File** ▸ **Save As** to save with a new filename).

In Ubuntu and Raspberry Pi, you'll need a program called GIMP (jump ahead to page 203 in Chapter 13 if you don't have this installed) —select **Image** ▸ **Scale Image** in GIMP, and then **File** ▸ **Export As** to save it with a new filename.

Import the `time` module and then use the `sleep` function (try with `time.sleep(0.5)`) to make the photos appear more slowly.

PART II

BOUNCE!

11
BEGINNING YOUR FIRST GAME: BOUNCE!

So far, we've covered the fundamentals of computer programming. You've learned how to use variables to store information, `if` statements for conditional code, and `for` loops for repeating code. You know how to create functions to reuse your code, and how to use classes and objects to divide your code into smaller chunks that are easier to understand. You've learned how to draw graphics on the screen with both the `turtle` and `tkinter` modules. Now, it's time to use that knowledge to create your first game.

WHACK THE BOUNCING BALL

We're going to develop a game with a bouncing ball and a paddle. The ball will fly around the screen, and the player will bounce it off the paddle. If the ball hits the bottom of the screen, the game ends. Figure 11-1 shows a preview of the finished game.

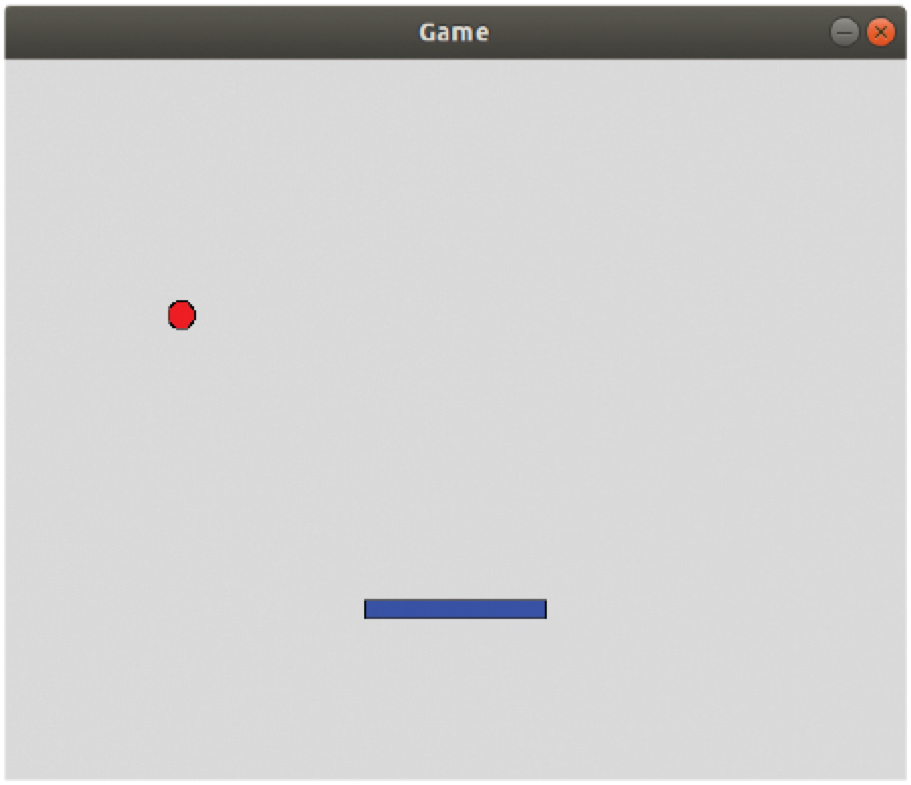


Figure 11-1: Bounce! game

Our game may look quite simple, but the code will be a bit trickier than what we've written so far because it needs to handle a lot of things. For example, it needs to animate the paddle and the ball, and detect when the ball hits the paddle or the walls.

In this chapter, we'll begin creating the game by adding a game canvas and a bouncing ball. In the next chapter, we'll complete the game by adding the paddle.

CREATING THE GAME CANVAS

To create your game, first open a new file in IDLE by choosing **File ▸ New File**. Then import `tkinter` and create a `canvas` to draw on:

```
  from tkinter import *
  import random
  import time
  tk = Tk()
❶ tk.title('Bounce Game')
```

```
tk.resizable(0, 0)
tk.wm_attributes('-topmost', 1)
canvas = Canvas(tk, width=500, height=400, bd=0, highlightthickness=0)
canvas.pack()
tk.update()
```

This code is a little different from previous examples. First, we import the `time` and `random` modules with `import random` and `import time` , to use a bit later in the code. The `random` module provides (among other things) functions for creating random numbers, and `time` has a useful function that will tell Python to pause what it's doing for a period of time.

With `tk.title('Bounce Game')` ❶ , we use the `title` function of the `tk` object we created with `tk = Tk()` to give the window a title. Then we use `resizable` to make the window a fixed size. The parameters (`0, 0`) say, "The size of the window cannot be changed either horizontally or vertically." Next, we call `wm_` attributes to tell `tkinter` to place the window containing our canvas in front of all other windows (`'-topmost'`).

When we create a `Canvas` object, we pass in a few more named parameters than with previous examples. For example, both `bd=0` and `highlightthickness=0` make sure there's no border around the outside of the canvas, which makes it look better on our game screen. The line `canvas.pack()` tells the canvas to size itself according to the width and height parameters given in the preceding line. Finally, `tk.update()` tells `tkinter` to initialize itself for the animation in our game. Without this last line, nothing would work as expected.

Make sure you save your code as you go. Give it a meaningful filename the first time you save it, such as *paddleball.py* .

CREATING THE BALL CLASS

Now we'll create the class for the ball. We'll begin with the code we need for the ball to draw itself on the canvas. We need to do the following:

1. Create a `class` called `Ball` that takes parameters for the canvas and the color of the ball we're going to draw.
2. Save the canvas as an object variable because we'll draw our ball on it.
3. Draw a filled circle on the canvas by using the value of the `color` parameter as the fill color.
4. Save the identifier `tkinter` returns when it draws the circle (oval) because we're going use this to move the ball around the screen.
5. Move the oval to the middle of the canvas.

This code should be added just after the first three lines in the file (after `import time`):

```
from tkinter import *
import random
import time

class Ball:
    def __init__(self, canvas, color):
        self.canvas = canvas
        self.id = canvas.create_oval(10, 10, 25, 25, fill=color)
        self.canvas.move(self.id, 245, 100)

    def draw(self):
        pass
```

First, we name our class `Ball` . Then we create an initialization function (as described in "Initializing an Object" on page 109) that takes the parameters `canvas` and `color` . We set the object variable `canvas` to the value of the parameter of the same name. We then call the `create_oval` function with five parameters: *x* and *y* coordinates for the top-left corner (10 and 10), *x* and *y* coordinates for the bottom-right corner (25 and 25), and the fill color for the oval.

The `create_oval` function returns an identifier for the shape it's drawn, which we store in the object variable `id` . We move the oval to the middle of the canvas (coordinates 245, 100). The canvas knows what to move because we use the stored shape identifier (`id`) to identify it.

On the last two lines of the `Ball` class, we create the `draw` function with `def draw(self)` , and the body of the function is simply the `pass` keyword. At the moment, it does nothing, but we'll add more to this function shortly.

Now that we've created our `Ball` class, we need to create an object of this class (remember: a class describes what it can do, but the object is the thing that actually does it). Add this code to the bottom of the program to create a red ball object:

```
ball = Ball(canvas, 'red')
```

You can run this program using **Run ▸ Run Module** . If you do this outside of IDLE, the canvas will appear for a split second and then vanish. To stop the window from closing immediately, we need to add an animation loop, which is called the *main loop* of our game. (IDLE already has a main loop, which is why the window doesn't vanish when you run it there.)

A main loop is the central part of a program that generally controls most of what it does. Our main loop, for the moment, just tells `tkinter` to redraw the screen. The loop, also called the *infinite loop* , keeps running forever (or at least until we close the window), constantly telling `tkinter` to redraw the screen and then sleeping for one hundredth of a second by using `time.sleep` . We'll add this code to the end of our program:

```
ball = Ball(canvas, 'red')

while True:
    tk.update_idletasks()
    tk.update()
    time.sleep(0.01)
```

Now if you run the code, the ball should appear in the center of the canvas, as shown in Figure 11-2 .

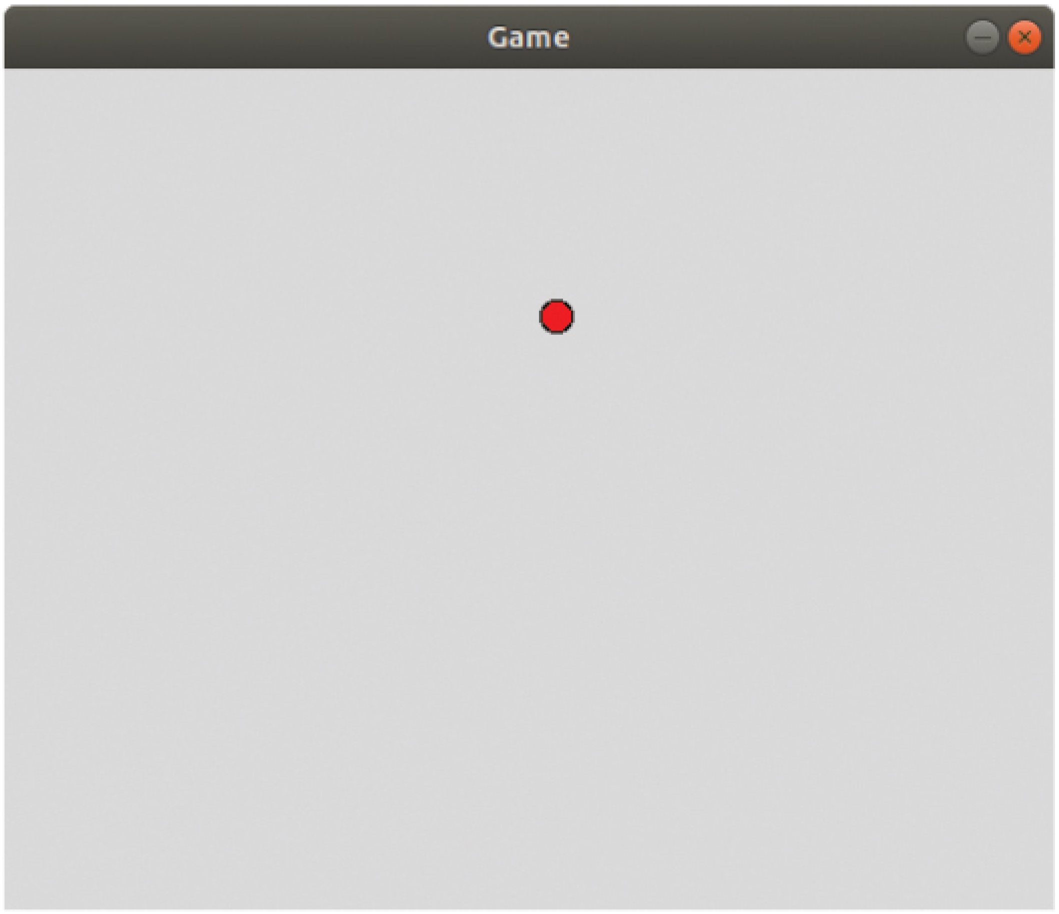
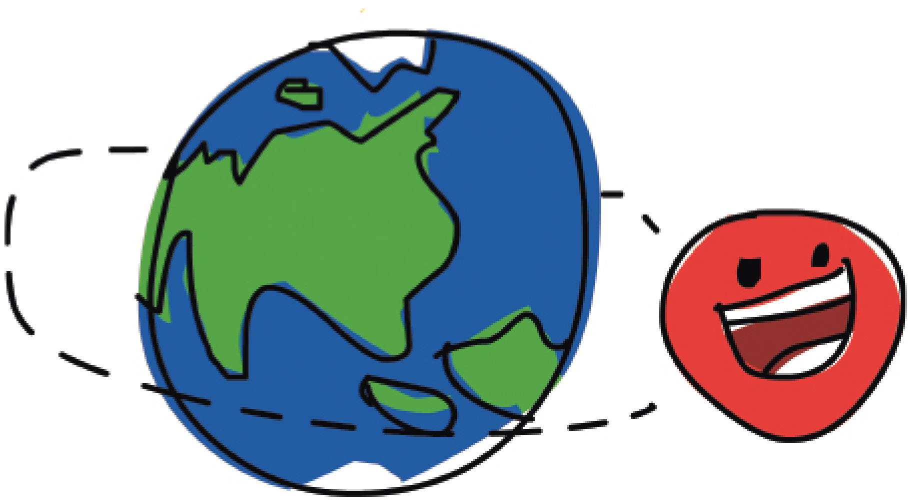


Figure 11-2: Ball in the center of the canvas

ADDING SOME ACTION

Now that we have the `Ball` class set up, it's time to animate the ball. We'll make it move, bounce, and change direction.



MAKING THE BALL MOVE

To move the ball, change the `draw` function as follows:

```
class Ball:
    def __init__(self, canvas, color):
        self.canvas = canvas
        self.id = canvas.create_oval(10, 10, 25, 25, fill=color)
        self.canvas.move(self.id, 245, 100)

    def draw(self):
        self.canvas.move(self.id, 0, -1)
```

Since `__init__` saved the `canvas` parameter as the `canvas` object variable, we use that variable with `self.canvas` and call the `move` function on the canvas.

We pass three parameters to `move` : the `id` of the oval, and the numbers `0` and `-1` . The `0` tells the ball to not move horizontally, and the `-1` tells the ball to move 1 pixel up the screen.

We're making this small change because it's beneficial to try things out as we go. Imagine writing the entire code for our game all at once and then discovering that it didn't work. Where would we begin looking to figure out what went wrong?

We'll also change the main loop at the bottom of our program. In the block of the `while` loop (our main loop), we add a call to the `ball` object's draw function, like so:

```
while True:
    ball.draw()
    tk.update_idletasks()
    tk.update()
    time.sleep(0.01)
```

If you run this code now, the ball should move up the canvas and vanish off the top of the screen—the commands update _idletasks and update tell tkinter to hurry up and draw what is on the canvas.

The time.sleep command is a call to the sleep function of the time module, which tells Python to sleep for one hundredth of a second (0.01). This ensures our program won't run so fast that the ball vanishes before you even see it.

This loop is basically saying, "Move the ball a little, redraw the screen with the new position, sleep for a moment, and then start over again."

NOTE

You may see error messages written to the Python Shell when you close the game window. This is because closing the window interrupts what tkinter *is doing, and Python is complaining about it. We can safely ignore these types of errors.*

Your game code should now look like this:

```
from tkinter import *
import random
import time

class Ball:
    def __init__(self, canvas, color):
        self.canvas = canvas
        self.id = canvas.create_oval(10, 10, 25, 25, fill=color)
        self.canvas.move(self.id, 245, 100)

    def draw(self):
        self.canvas.move(self.id, 0, -1)
```

```
tk = Tk()
tk.title('Bounce Game')
tk.resizable(0, 0)
tk.wm_attributes('-topmost', 1)
canvas = Canvas(tk, width=500, height=400, bd=0, highlightthickness=0)
canvas.pack()
tk.update()

ball = Ball(canvas, 'red')

while True:
    ball.draw()
    tk.update_idletasks()
    tk.update()
    time.sleep(0.01)
```

If you run this code, the ball will start moving upward and sail off the top of the window.

MAKING THE BALL BOUNCE

A ball that vanishes off the top of the screen isn't particularly useful for our game, so let's make it bounce. First, we'll save a few additional object variables in the initialization function of the `Ball` class, like so:

```
    def __init__(self, canvas, color):
        self.canvas = canvas
        self.id = canvas.create_oval(10, 10, 25, 25, fill=color)
        self.canvas.move(self.id, 245, 100)
        self.x = 0
        self.y = -1
        self.canvas_height = self.canvas.winfo_height()
```

We've added three more lines to our program. With `self.x = 0`, we set the object variable `x` to 0; with `self.y = -1`, we set the variable `y` to −1. Lastly, we set the object variable `canvas_height` by calling the `winfo_height` canvas function. This function returns the current height of the canvas.

Next, we change the `draw` function again:

```
    def draw(self):
        self.canvas.move(self.id, self.x, self.y)
        pos = self.canvas.coords(self.id)
        if pos[1] <= 0:
            self.y = 1
        if pos[3] >= self.canvas_height:
            self.y = -1
```

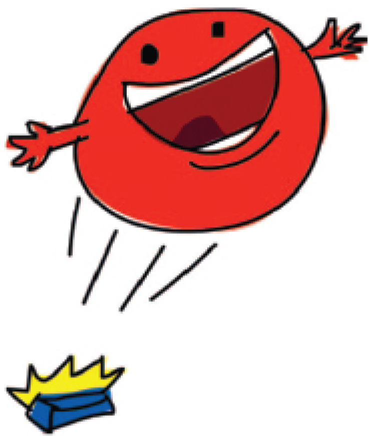
We first change the call to the canvas's `move` function by passing the `x` and `y` object variables. Next, we create a variable called `pos` by calling the `coords` canvas function. This function returns the current *x* and *y* coordinates of anything drawn on the canvas as long as you know its identifying number. In this case, we pass `coords` the object variable `id` , which contains the oval's identifier.

The `coords` function returns the coordinates as a list of four numbers. If we print the results of calling this function, we'll see something like this:

```
print(self.canvas.coords(self.id))
[255.0, 29.0, 270.0, 44.0]
```

The first two numbers in the list (`255.0` and `29.0`) contain the top-left coordinates of the oval (*x1* and *y1*), and the second pair (`270.0` and `44.0`) are the bottom-right *x2* and *y2* coordinates. We'll use these values in the next few lines of code.

We continue our code by seeing if the *y1* coordinate (that's the top of the ball!) is less than or equal to 0. If so, we set the `y` object variable to `1` . In effect, we're saying if you hit the top of the screen, `tkinter` will stop subtracting 1 from the vertical position, and the ball will stop moving up (this is a simple version of *collision detection*).

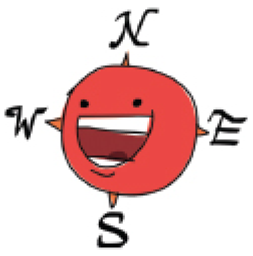


We then see if the *y2* coordinate (that's the bottom of the ball!) is greater than or equal to the variable `canvas_height` . If it is, we set the `y` object variable back to `-1` . Now the ball will stop moving down and head back up again.

Run this code now, and the ball should bounce up and down the canvas until you close the window.

CHANGING THE BALL'S STARTING DIRECTION

Making a ball bounce slowly up and down isn't much of a game, so let's enhance things a bit by changing the ball's starting direction—the angle it moves when the game starts.



In the `__init__` function, change these two lines:

```
    self.x = 0
    self.y = -1
```

to the following code (make sure you have the right number of spaces—there are eight—at the beginning of each line):

```
    starts = [-3, -2, -1, 1, 2, 3]
    self.x = random.choice(starts)
    self.y = 3
```

We begin by creating the `starts` variable with a list of six numbers. Then we set the value of the `x` variable, using the `random.choice` function, which returns a random item from a list. By using that function, `x` can be any number in the list, from –3 to 3.

Lastly, we change `y` to –3 (so the ball starts the game moving upward). Now our ball can move in any direction, but we need to make a few more additions to be sure it won't vanish off the side of the screen.

Add the following line to the end of the __init__ function to save the width of the canvas to a new canvas_width object variable:

```
    self.canvas_width = self.canvas.winfo_width()
```

We'll use this new object variable in the draw function to see if the ball has hit the left or right side of the canvas:

```
    if pos[0] <= 0 or pos[2] >= self.canvas_width:
        self.x = self.x * -1
```

If the leftmost position of the ball is less than or equal to 0, or the rightmost position of the ball is greater than or equal to the width of the canvas, we do this odd little calculation self.x = self.x * -1 . The x variable is set to the current value of x multiplied by –1. So if the value of x is 2, the new value will be –2. If the value of x is –3, the new value will be 3. So when the ball hits a side, it will bounce back in the opposite direction. We can do a similar check for the top and bottom of the canvas, using the canvas height and multiplying the y variable by –1. Your draw function should now look like this:

```
    def draw(self):
        self.canvas.move(self.id, self.x, self.y)
        pos = self.canvas.coords(self.id)
        if pos[0] <= 0 or pos[2] >= self.canvas_width:
            self.x = self.x * -1
        if pos[1] <= 0 or pos[3] >= self.canvas_height:
            self.y = self.y * -1
```

The full program should look like this:

```
from tkinter import *
import random
import time

class Ball:
    def __init__(self, canvas, color):
        self.canvas = canvas
        self.id = canvas.create_oval(10, 10, 25, 25, fill=color)
        self.canvas.move(self.id, 245, 100)
        starts = [-3, -2, -1, 1, 2, 3]
        self.x = random.choice(starts)
        self.y = -3
        self.canvas_height = self.canvas.winfo_height()
        self.canvas_width = self.canvas.winfo_width()

    def draw(self):
        self.canvas.move(self.id, self.x, self.y)
        pos = self.canvas.coords(self.id)
        if pos[0] <= 0 or pos[2] >= self.canvas_width:
            self.x = self.x * -1
        if pos[1] <= 0 or pos[3] >= self.canvas_height:
            self.y = self.y * -1

tk = Tk()
tk.title('Bounce Game')
tk.resizable(0, 0)
tk.wm_attributes('-topmost', 1)
canvas = Canvas(tk, width=500, height=400, bd=0, highlightthickness=0)
canvas.pack()
tk.update()

ball = Ball(canvas, 'red')

while True:
    ball.draw()
    tk.update_idletasks()
    tk.update()
    time.sleep(0.01)
```

Save and run the code, and the ball should bounce around the screen without vanishing.

WHAT YOU LEARNED

In this chapter, we started creating our first game using the `tkinter` module. We created a `Ball` object and animated it to move around the screen. We used coordinates to check when the ball hits the sides of the canvas so we can make it bounce. We also used the `choice` function in the `random` module so our ball doesn't always start moving in the same direction. In the next chapter, we'll complete the game by adding the paddle.

PROGRAMMING PUZZLES

#1: CHANGING COLORS

Try changing the starting color of the ball and the background color of the canvas—try a few different combinations of colors and see which ones you like.

#2: FLASHING COLORS

Because there's a loop at the bottom of our code, it should be quite easy to change the color of the ball as it moves across the screen. We can add some code to the loop that picks different colors (think about the `choice` function we used earlier in the chapter), and then updates the color of the ball (perhaps by calling a new function on our `Ball` class). To do this, you'll need to use the `itemconfig` function on the canvas (see "More Ways to Use the Identifier" on page 165).

#3: TAKE YOUR POSITIONS!

Try to change the code so the ball starts in a different position on the screen. You could make the position random by using the `random` module (see section "Drawing a Lot of Rectangles" on page 145 for an example of how to use the `randrange` function in that module). But you'll have to ensure the ball doesn't start too close, or below, the paddle, which will make the game impossible to play.

#4: ADDING THE PADDLE . . . ?

Based on the code we've created so far, can you figure out how to add the paddle before reaching the next chapter? If you look back at Chapter 10 , you might be able to figure out how to draw it before moving on. Then check the next few pages to see if you got it right!

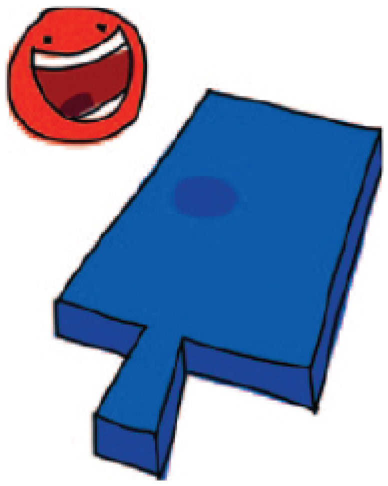
12
FINISHING YOUR FIRST GAME: BOUNCE!

In the previous chapter, we got started building our first game, *Bounce* !, by creating a canvas and adding a bouncing ball to our code. Right now, our ball will bounce around the screen forever, which doesn't make for much of a game. In this chapter, we'll add a paddle for the player to use. We'll also add an element of chance to the game, which will make it more challenging and fun to play.

ADDING THE PADDLE

There's not much fun to be had with a bouncing ball when there's nothing to hit it with. So let's create a paddle!

We begin by adding the following code directly after the `Ball` class to create a paddle (you'll put this in a new line below the ball's `draw` function):



```
class Paddle:
    def __init__(self, canvas, color):
        self.canvas = canvas
        self.id = canvas.create_rectangle(0, 0, 100, 10, fill=color)
        self.canvas.move(self.id, 200, 300)

    def draw(self):
        pass
```

This added code is almost exactly the same as what we did with our first version of the `Ball` class, except that we call `create_rectangle` (rather than `create_oval`) and we move the rectangle to position 200, 300 (200 pixels across and 300 pixels down).

Next, at the bottom of your code listing, create an object of the `Paddle` class, and then change the main loop to call the paddle's `draw` function, as shown here:

```
❶ paddle = Paddle(canvas, 'blue')
  ball = Ball(canvas, 'red')

  while True:
      ball.draw()
   ❷ paddle.draw()
      tk.update_idletasks()
      tk.update()
      time.sleep(0.01)
```

Changes can be seen at ❶ and ❷ . If you run the game now, you should see the bouncing ball and a stationary rectangular paddle (

Figure 12-1).

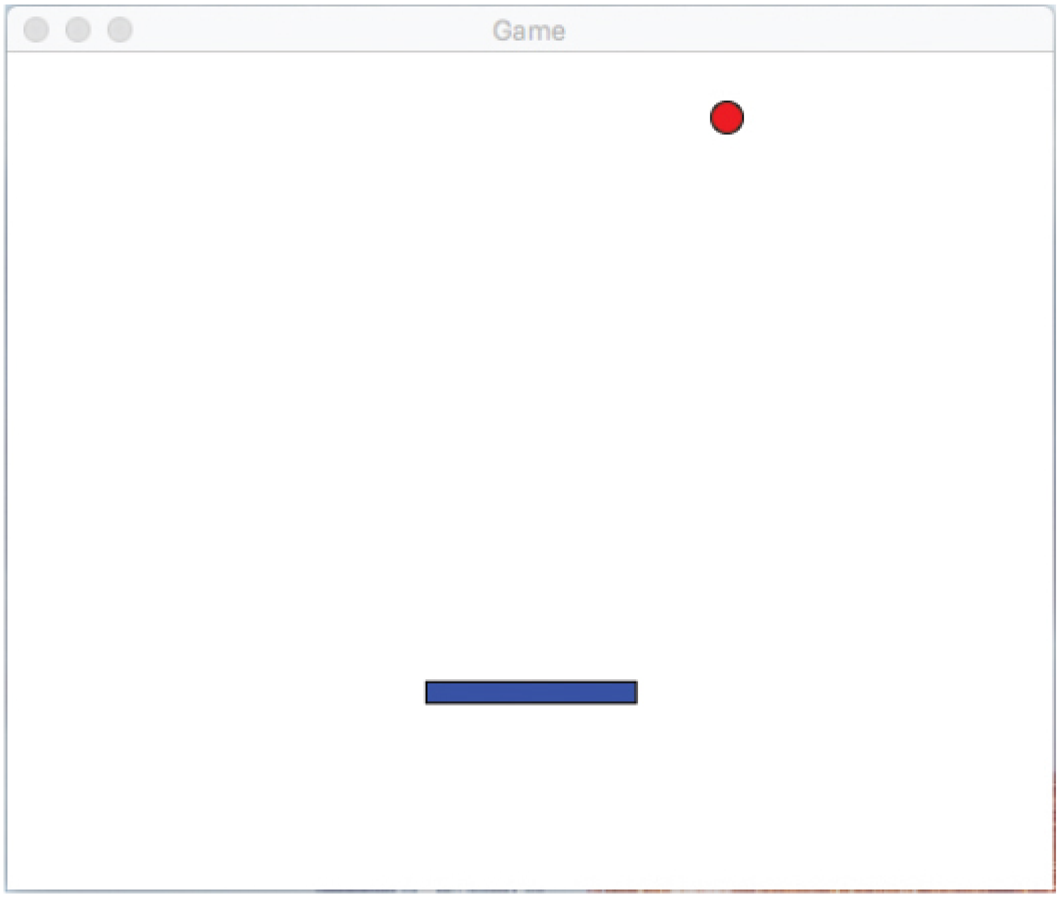
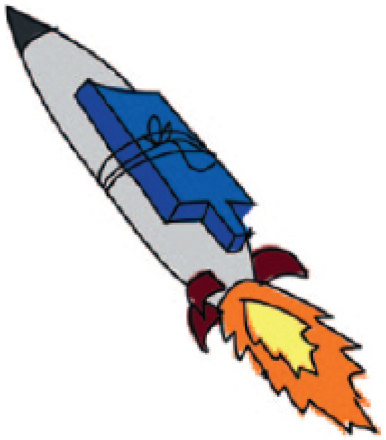


Figure 12-1: Ball and paddle

MAKING THE PADDLE MOVE

To make the paddle move left and right, we'll use event bindings to bind the *left* and *right* arrow keys to new functions in the `Paddle` class. When the player presses the left arrow key, the `x` variable will be set to –2 (to move left). Pressing the right arrow key sets the `x` variable to 2 (to move right).



The first step is to add the `x` object variable to the `__init__` function of our `Paddle` class, as well as a variable for the canvas width, as we did with the `Ball` class:

```
    def __init__(self, canvas, color):
        self.canvas = canvas
        self.id = canvas.create_rectangle(0, 0, 100, 10, fill=color)
        self.canvas.move(self.id, 200, 300)
      ❶ self.x = 0
      ❷ self.canvas_width = self.canvas.winfo_width()
```

See ❶ and ❷ for changes. Now we'll add the functions for changing the direction between left (`turn_left`) and right (`turn_right`) just after the `draw` function:

```
    def turn_left(self, evt):
        self.x = -2

    def turn_right(self, evt):
        self.x = 2
```

We can bind these functions to the correct key in the `__init__` function of the class with these two lines. We used binding in "Making an Object React to Something" on page 162 to make Python call a function when a key is pressed. In this case, we bind the `turn_left` function of our `Paddle` class to the left arrow key, using the event name `<KeyPress-Left>` . We then bind the `turn_right` function to the right arrow key, using the event name `<KeyPress-Right>` . Our `__init__` function now looks like this:

```
    def __init__(self, canvas, color):
        self.canvas = canvas
        self.id = canvas.create_rectangle(0, 0, 100, 10, fill=color)
        self.canvas.move(self.id, 200, 300)
        self.x = 0
        self.canvas_width = self.canvas.winfo_width()
     ❶ self.canvas.bind_all('<KeyPress-Left>', self.turn_left)
     ❷ self.canvas.bind_all('<KeyPress-Right>', self.turn_right)
```

See ❶ and ❷ for the changes. The `draw` function for the `Paddle` class is similar to that for the `Ball` class:

```
    def draw(self):
        self.canvas.move(self.id, self.x, 0)
        pos = self.canvas.coords(self.id)
        if pos[0] <= 0 or pos[2] >= self.canvas_width:
            self.x = 0
```

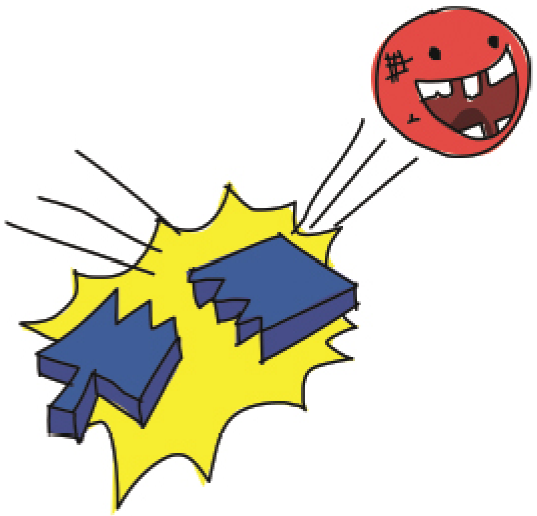
We use the `canvas` 's `move` function to move the paddle in the direction of the `x` variable with `self.canvas.move(self.id, self.x, 0)` . Then we get the paddle's coordinates to see if it has hit the left or right side of the screen, using the value in `pos` . Rather than bouncing like the ball, the paddle should stop moving. So, when the left *x* coordinate (`pos[0]`) is less than or equal to 0 (`<= 0`), we set the `x` variable to 0 with `self.x = 0` . In the same way, when the right *x* coordinate (`pos[2]`) is greater than or equal to the canvas width (`>= self.canvas_width`), we also set the `x` variable to 0.

Note

If you run the program now, you may need to click the canvas before the game will recognize the left and right arrow key actions. Clicking the canvas gives the canvas focus, which means it knows to take charge when someone presses a key on the keyboard.

FINDING OUT WHEN THE BALL HITS THE PADDLE

At this point in our code, the ball won't hit the paddle. In fact, the ball will fly straight through the paddle. The ball needs to know when it has hit the paddle, just as it needs to know when it's hit a wall.



We could solve this problem by adding code to the `draw` function (where we have code that checks for walls), but it's a better idea to move this code into new functions to break things into smaller chunks. If we put too much code in one place (inside one function, for example), we can make the code much more difficult to understand. Let's make the necessary changes.

First, we change the ball's `__init__` function so we can pass in the `paddle` object as a parameter:

```
class Ball:
  ❶ def __init__(self, canvas, paddle, color):
        self.canvas = canvas
      ❷ self.paddle = paddle
        self.id = canvas.create_oval(10, 10, 25, 25, fill=color)
        self.canvas.move(self.id, 245, 100)
        starts = [-3, -2, -1, 1, 2, 3]
        self.x = random.choice(starts)
        self.y = -3
        self.canvas_height = self.canvas.winfo_height()
        self.canvas_width = self.canvas.winfo_width()
```

Notice that we change the parameters of `__init__` to include the paddle ❶ . Then, we assign the paddle parameter to the object variable `paddle` ❷ .

Having saved the `paddle` object, we need to change the code where we create the `ball` object. This change is at the bottom of our program, just before the main `while` loop:

```
paddle = Paddle(canvas, 'blue')
ball = Ball(canvas, paddle, 'red')

while True:
    ball.draw()
    paddle.draw()
    tk.update_idletasks()
    tk.update()
    time.sleep(0.01)
```

To see if the ball has struck the paddle, we need code that's a little more complicated than the previously added code to check for walls. We'll name this function `hit_paddle` and call it in the `draw` function of the `Ball` class, where we check to see if the ball has hit the bottom of the screen:

```
    def draw(self):
        self.canvas.move(self.id, self.x, self.y)
        pos = self.canvas.coords(self.id)
        if pos[1] <= 0 or pos[3] >= self.canvas_height:
            self.y = self.y * -1
      ❶ if self.hit_paddle(pos) == True:
          ❷ self.y = self.y * -1
        if pos[0] <= 0 or pos[2] >= self.canvas_width:
            self.x = self.x * -1
```

In the newly added code, if `hit_paddle` returns `True` ❶ , we change the direction of the ball by setting the `y` object variable to its value multiplied by –1 ❷ (the same as when it hits the top or bottom of the canvas). With this code, we're basically saying, "If the ball (`self`) has hit the paddle, then we reverse its vertical direction."

We could combine the top, bottom, and paddle checks into one `if` - statement—but it's easier for new programmers to read this code if we keep them separate.

Don't try to run the game yet; we still need to create the `hit_paddle` function. Let's do that now. Add the `hit_paddle` function just before the `draw` function in the `Ball` class:

```
    def hit_paddle(self, pos):
        paddle_pos = self.canvas.coords(self.paddle.id)
        if pos[2] >= paddle_pos[0] and pos[0] <= paddle_pos[2]:
            if pos[3] >= paddle_pos[1] and pos[3] <= paddle_pos[3]:
                return True
        return False
```

First, we define the function with the `pos` parameter. This parameter contains the ball's current coordinates. Then, we get the paddle's coordinates and store them in the `paddle_pos` variable.

Next, we have the first part of our `if-then` statement, which says, "If the right side of the ball is greater than the left side of the paddle, and the left side of the ball is less than the right side of the paddle. . ." Here, `pos[2]` contains the *x* coordinate for the ball's right side, and `pos[0]` contains the *x* coordinate for its left side. The variable `paddle_pos[0]` contains the *x* coordinate for the paddle's left side, and `paddle_pos[2]` contains its *x* coordinate for the right side. Figure 12-2 shows how these coordinates look when the ball is about to hit the paddle.

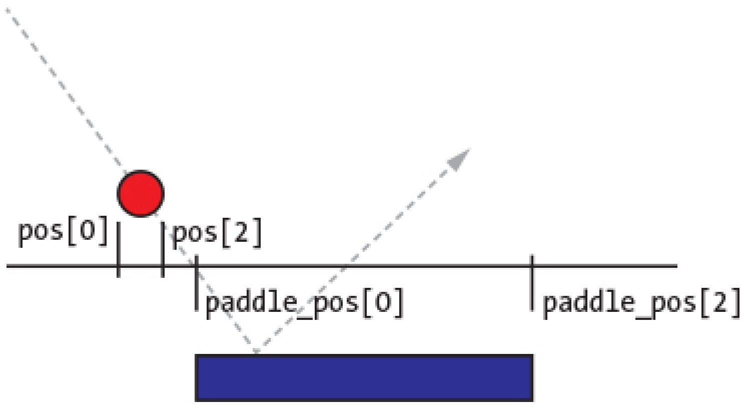


Figure 12-2: Ball about to hit the paddle—showing horizontal coordinates

The ball is falling toward the paddle, but in this case, you see that the right side of the ball (`pos[2]`) hasn't yet crossed over the left side of the paddle (that's `paddle_pos[0]`).

Next, we see if the bottom of the ball (`pos[3]`) is between the paddle's top (`paddle_pos[1]`) and bottom (`paddle_pos[3]`). In Figure 12-3 , you can see that the bottom of the ball (`pos[3]`) has yet to hit the top of the paddle (`paddle_pos[1]`).

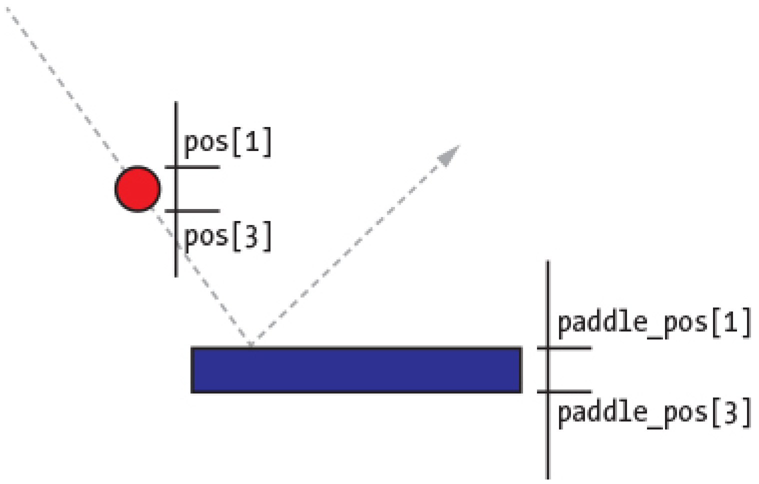


Figure 12-3: Ball about to hit the paddle—showing vertical coordinates

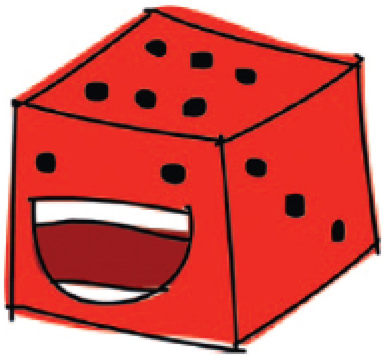
So, based on the current position of the ball, the `hit_paddle` function would return `False` .

Note

Why do we need to see if the bottom of the ball is between the top and bottom of the paddle? Why not just see if the bottom of the ball has hit the top of the paddle? Because each time we move the ball across the canvas, we move in 3-pixel jumps. If we just checked to see if the ball had reached the top of the paddle (`pos[1]`), we might have jumped past that position. In that case, the ball would continue traveling and would pass through the paddle without stopping.

ADDING AN ELEMENT OF CHANCE

Now it's time to turn our program into a game rather than just a bouncing ball and a paddle. Games need an *element of chance* , or a way for the player to lose. In our current game, the ball will bounce forever, so there's nothing to lose.



We'll finish our game by adding code that says that the game ends if the ball hits the bottom of the canvas (in other words, once it hits the ground).

First, we add the `hit_bottom` object variable to the bottom of the `Ball` class's `__init__` function:

```
        self.canvas_height = self.canvas.winfo_height()
        self.canvas_width = self.canvas.winfo_width()
        self.hit_bottom = False
```

Then we change the main loop at the bottom of the program, like this:

```
while True:
  ❶ if ball.hit_bottom == False:
        ball.draw()
        paddle.draw()
    tk.update_idletasks()
    tk.update()
    time.sleep(0.01)
```

Now the loop keeps checking `hit_bottom` ❶ to see if the ball has indeed hit the bottom of the screen. The code should continue moving the ball and paddle only if the ball hasn't touched the bottom, as you can see in the preceding `if` statement. The game ends when the ball and paddle stop moving. (We no longer animate them.)

The final change is to the `draw` function of the `Ball` class:

```
    def draw(self):
        self.canvas.move(self.id, self.x, self.y)
        pos = self.canvas.coords(self.id)
        if pos[1] <= 0:
            self.y = self.y * -1
     ❶ if pos[3] >= self.canvas_height:
            self.hit_bottom = True
        if self.hit_paddle(pos) == True:
            self.y = self.y * -1
        if pos[0] <= 0 or pos[2] >= self.canvas_width:
            self.y = self.y * -1
```

We altered the `if` statement to see if the ball has hit the bottom of the screen (that is, if the ball's position is greater than or equal to `canvas_height`) ❶ . If so, in the following line, we set `hit_bottom` to `True` , rather than changing the value of the `y` variable, because there's no need to bounce the ball once it hits the bottom of the screen.

When you run the game now and miss hitting the ball with the paddle, all movement on your screen should stop. The game should end once the ball touches the bottom of the canvas, as shown in Figure 12-4
.

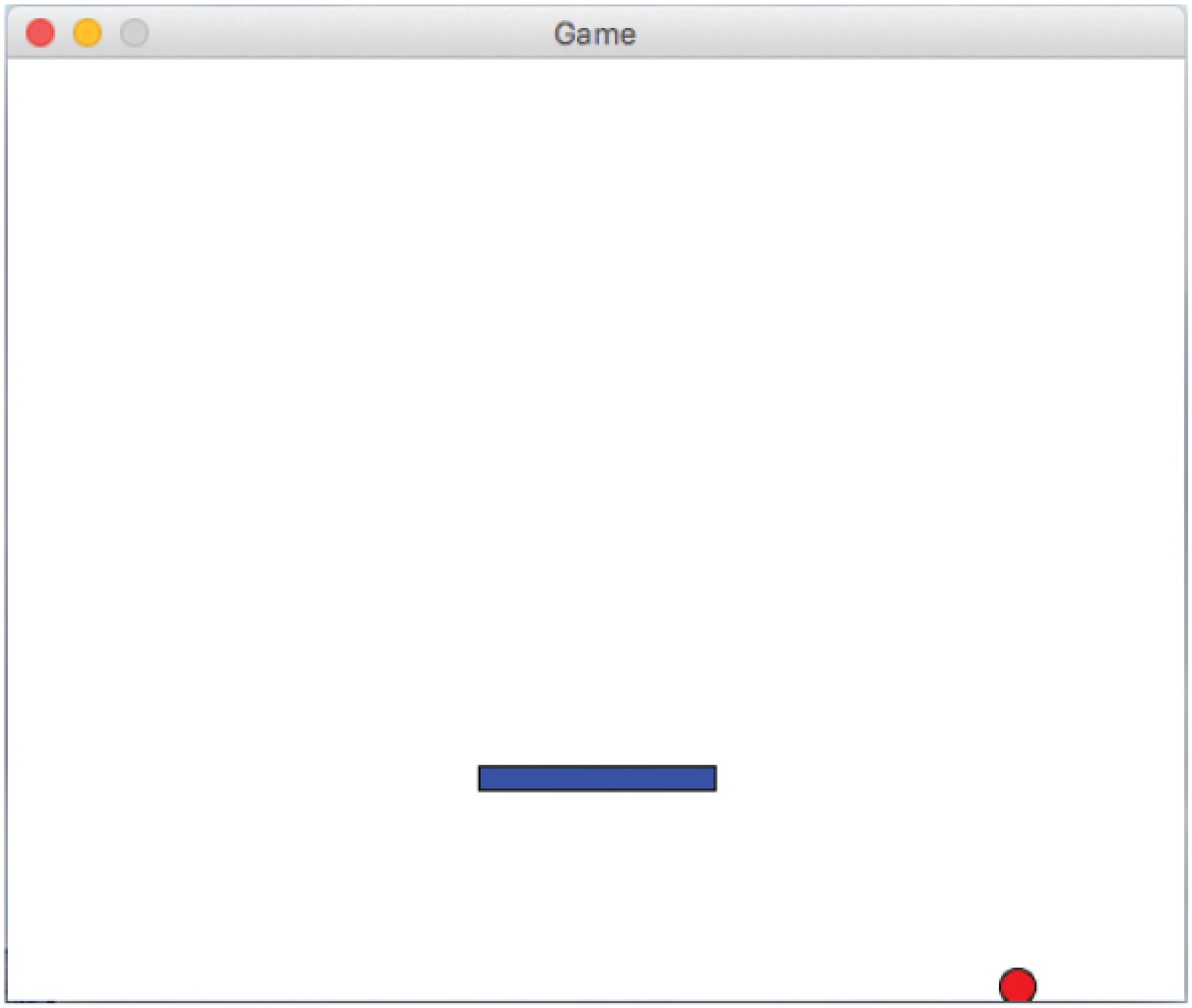


Figure 12-4: Ball hitting the bottom of the screen

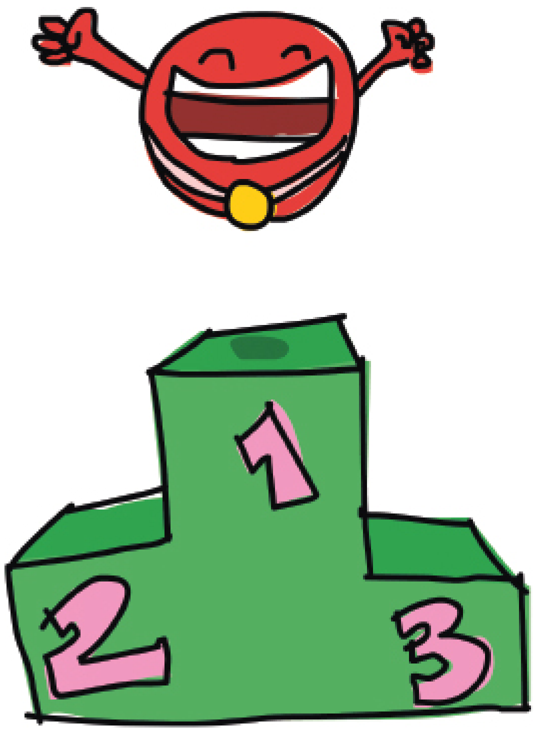
Your program should now look like the following code. If you have trouble getting your game to work, check what you've entered against this code:

```
from tkinter import *
import random
import time

class Ball:
    def __init__(self, canvas, paddle, color):
        self.canvas = canvas
        self.paddle = paddle
        self.id = canvas.create_oval(10, 10, 25, 25, fill=color)
        self.canvas.move(self.id, 245, 100)
        starts = [-3, -2, -1, 1, 2, 3]
        self.x = random.choice(starts)
        self.y = -3
        self.canvas_height = self.canvas.winfo_height()
        self.canvas_width = self.canvas.winfo_width()
        self.hit_bottom = False

    def hit_paddle(self, pos):
        paddle_pos = self.canvas.coords(self.paddle.id)
        if pos[2] >= paddle_pos[0] and pos[0] <= paddle_pos[2]:
            if pos[3] >= paddle_pos[1] and pos[3] <= paddle_pos[3]:
                return True
        return False

    def draw(self):
        self.canvas.move(self.id, self.x, self.y)
        pos = self.canvas.coords(self.id)
        if pos[1] <= 0:
            self.y = self.y * -1
        if pos[3] >= self.canvas_height:
            self.hit_bottom = True
        if self.hit_paddle(pos) == True:
            self.y = self.y * -1
        if pos[0] <= 0 or pos[2] >= self.canvas_width:
            self.y = self.y * -1

class Paddle:
    def __init__(self, canvas, color):
        self.canvas = canvas
        self.id = canvas.create_rectangle(0, 0, 100, 10, fill=color)
        self.canvas.move(self.id, 200, 300)
        self.x = 0
        self.canvas_width = self.canvas.winfo_width()
        self.canvas.bind_all('<KeyPress-Left>', self.turn_left)
        self.canvas.bind_all('<KeyPress-Right>', self.turn_right)

    def draw(self):
        self.canvas.move(self.id, self.x, 0)
        pos = self.canvas.coords(self.id)
        if pos[0] <= 0 or pos[2] >= self.canvas_width:
            self.x = 0

    def turn_left(self, evt):
        self.x = -2

    def turn_right(self, evt):
        self.x = 2

tk = Tk()
tk.title('Bounce Game')
tk.resizable(0, 0)
tk.wm_attributes('-topmost', 1)
```

```
canvas = Canvas(tk, width=500, height=400, bd=0, highlightthickness=0)
canvas.pack()
tk.update()

paddle = Paddle(canvas, 'blue')
ball = Ball(canvas, paddle, 'red')

while True:
    if ball.hit_bottom == False:
        ball.draw()
        paddle.draw()
    tk.update_idletasks()
    tk.update()
    time.sleep(0.01)
```

WHAT YOU LEARNED

In this chapter, we finished creating our first game using the `tkinter` module. We created classes for the paddle used in our game, and used coordinates to check when the ball hits the paddle or the walls of our game canvas. We used event bindings to bind the left and right arrow keys to the movement of the paddle, and used a main loop to call the `draw` function, to animate it. Finally, we changed our code to add an element of chance, so that when the player misses the ball and it hits the bottom of the canvas, the game is over.



PROGRAMMING PUZZLES

At the moment, our game is pretty simple. You could change a lot to create a more interesting game. Try enhancing your code in the following ways, and then compare your answers with the solutions at *http://python-for-kids.com* .

#1: DELAY THE GAME START

Our game starts quickly, and you need to click the canvas before it will recognize pressing the left and right arrow keys on your keyboard. Can you add a delay to the start of the game in order to give the player enough time to click the canvas? Or even better, can you add an event binding for a mouse click, which starts the game only then?

Hint 1: You've already added event bindings to the `Paddle` class, so that might be a good place to start.

Hint 2: The event binding for the left mouse button is the string `'<Button-1>'` .

#2: A PROPER "GAME OVER"

Everything freezes when the game ends, which isn't very player-friendly. Try adding the text "Game Over" when the ball hits the bottom of the screen. You can use the `create_text` function, but you might also find the named parameter `state` useful (it takes values such as `normal` and `hidden`). Have a look at `itemconfig` in "More Ways to Use the Identifier" on page 165 . As an additional challenge, add a delay so that the text doesn't appear right away.

#3: ACCELERATE THE BALL

In tennis, when a ball hits your racket, it sometimes flies away faster than the speed at which it arrived, depending on how hard you swing. The ball in our game goes at the same speed, whether or not the paddle is moving. Try changing the program so that the paddle's speed is passed on to the speed of the ball.

#4: RECORD THE PLAYER'S SCORE

How about recording the score? Every time the ball hits the paddle, the score should increase. Try displaying the score at the top-right corner of the canvas. You might want to look back at `itemconfig` in "More Ways to Use the Identifier" on page 165 for a hint.

PART III

MR. STICK MAN RACES FOR THE EXIT

13
CREATING GRAPHICS FOR THE MR. STICK MAN GAME

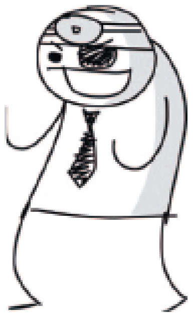
It's a good idea to develop a plan when creating a game (or any program). Your plan should include a description of what the game is about, as well as major elements and characters. When it's time to start programming, your description will help keep you focused on what you are trying to develop. Your game might not turn out exactly like the original description—this is to be expected! In this chapter, we'll begin developing a fun game called *Mr. Stick Man Races for the Exit* .

MR. STICK MAN GAME PLAN

Here's the description of our new game:

- Secret agent Mr. Stick Man is trapped in the lair of Dr. Innocuous; you must help him escape through the exit on the top floor.
- There is a stick figure that can run from left to right and jump up. Each floor has platforms that he must jump to.

- The goal is to reach the door to the exit before the game ends.



Based on this description, we'll need several images for Mr. Stick Man, the platforms, and the door. We'll write code to pull all this together, but before we get there, we'll create the graphics for our game. That way, we'll have something to work with in the next chapter.

We could draw these elements using graphics like our *Bounce!* game, but those are a bit too simple. Instead, we're going to create sprites.

Sprites are the moving objects in a game—typically a character of some kind. Sprites are usually *pre-rendered* , meaning they're drawn in advance (before the program runs) rather than created by the program itself using polygons, as in our *Bounce!* game. In this game, Mr. Stick Man, the platforms, and the door will be sprites. To create these images, we'll need to install a graphics program.

GETTING GIMP

There are several graphics programs available, but for this game, we need one that supports *transparency* (sometimes called the *alpha channel*), which lets images have sections where no colors are drawn on the screen. We need images with transparent parts because when one image passes over or near another as it moves across the screen, we don't want the background of one image to wipe out part of another. For example, in Figure 13-1 , the checkerboard pattern in the background represents the transparent area.

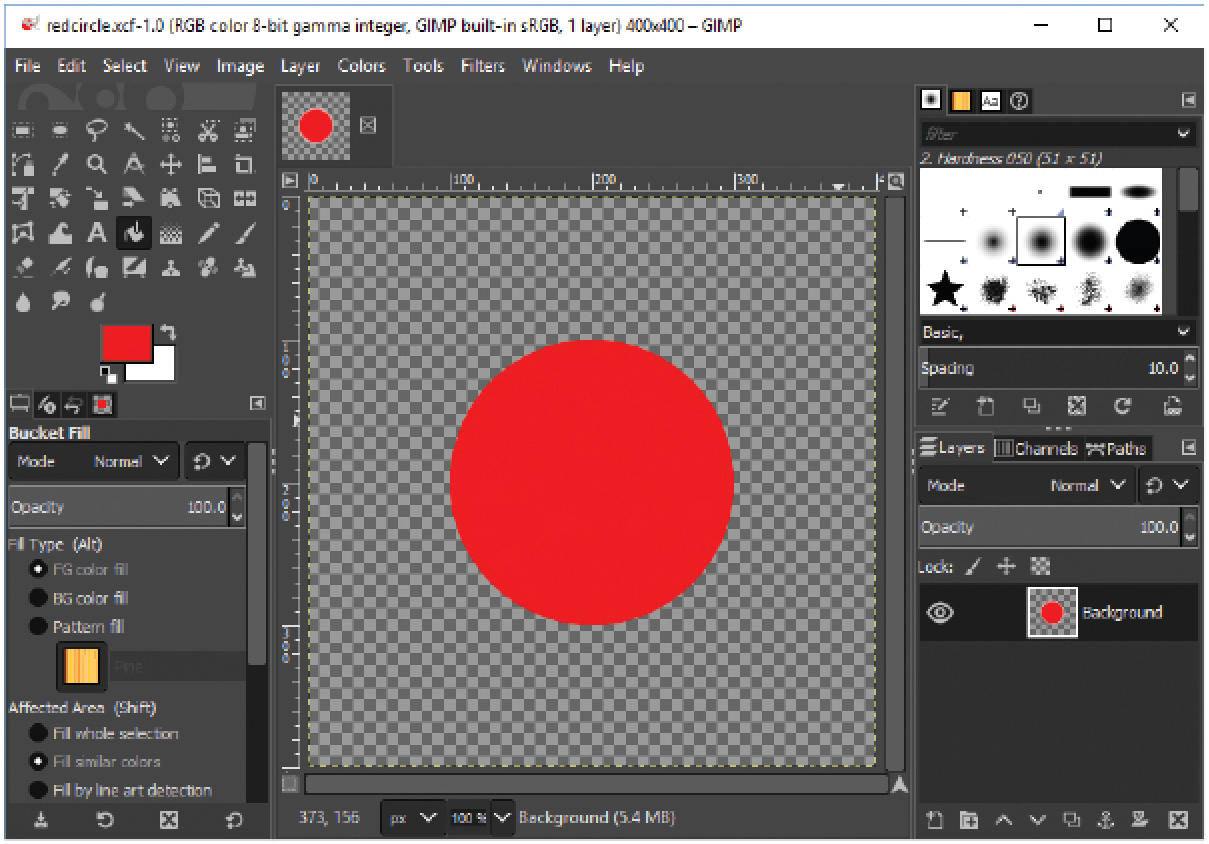
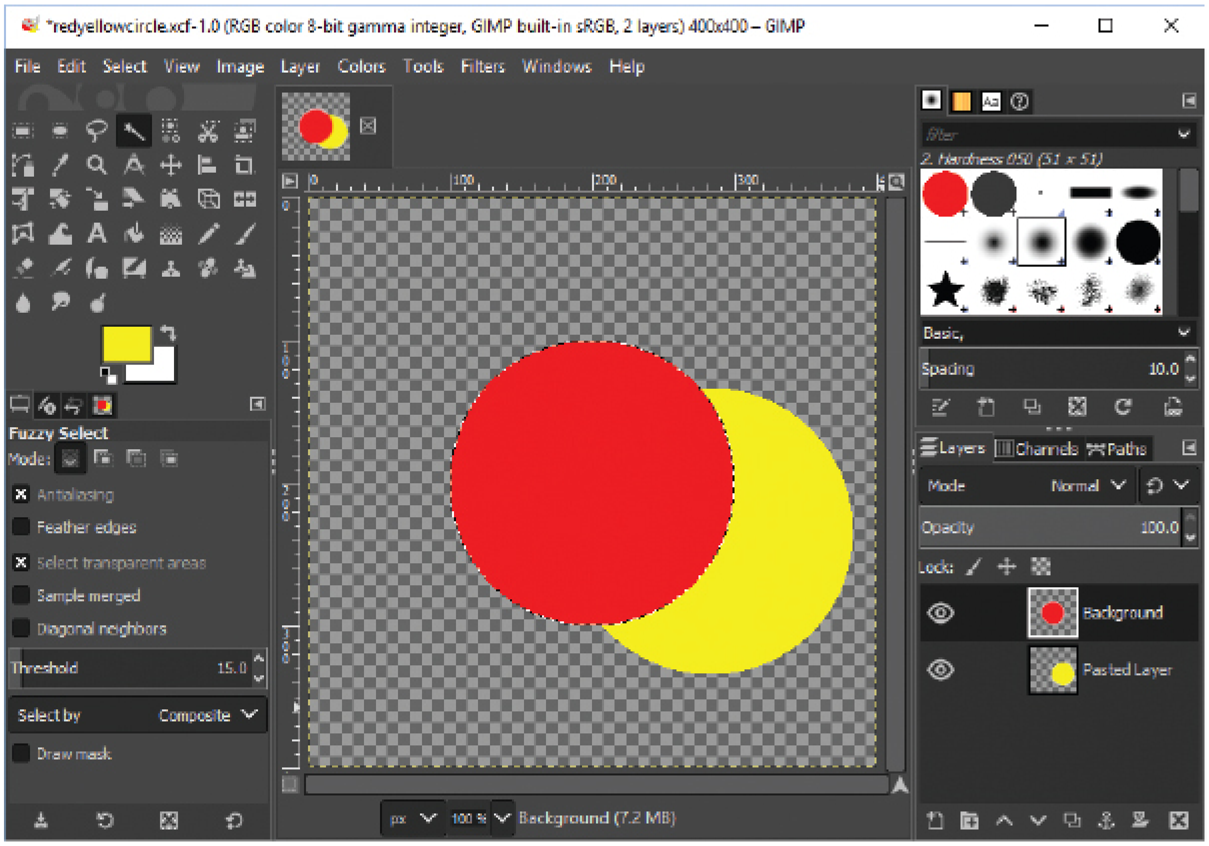

redcircle.xcf-1.0 (RGB color 8-bit gamma integer, GIMP built-in sRGB, 1 layer) 400x400 – GIMP
File Edit Select View Image Layer Colors Tools Filters Windows Help
Bucket Fill
Mode Normal
Opacity 100.0
Fill Type (Alt)
FG color fill
BG color fill
Pattern fill
Affected Area (Shift)
Fill whole selection
Fill similar colors
Fill by line art detection
2. Hardness 050 (51 × 51)
Basic,
Spacing 10.0
Layers Channels Paths
Mode Normal
Opacity 100.0
Lock:
Background
373, 156 px 100 % Background (5.4 MB)

Figure 13-1: Transparent background in GIMP

If we copy the entire image and paste it over the top of another image (also called *overlaying*), the background won't wipe anything out. This is shown in Figure 13-2 .



*redyellowcircle.xcf-1.0 (RGB color 8-bit gamma integer, GIMP built-in sRGB, 2 layers) 400x400 – GIMP

Fuzzy Select
Mode:
[✓] Antialiasing
[ ] Feather edges
[✓] Select transparent areas
[ ] Sample merged
[ ] Diagonal neighbors
Threshold 15.0
Select by Composite
[ ] Draw mask

2. Hardness 050 (51 × 51)
Basic,
Spacing 10.0

Mode Normal
Opacity 100.0
Lock:
Background
Pasted Layer

px 100 % Background (7.2 MB)

Figure 13-2: Overlaying images

GIMP (*http://www.gimp.org*), short for *GNU Image Manipulation Program* , is a free graphics program for Linux, macOS, and Windows that supports transparent images. Download and install it as follows.

- If you're using Windows or macOS, you'll find installers on the GIMP project page at *https://www.gimp.org/downloads/* .
- If you're using Ubuntu, install GIMP by opening the Ubuntu Software Center and entering *gimp* in the search box. Click the Install button for the "GNU Image Manipulation Program" when it appears in the results.
- If you're using Raspberry Pi, it's easiest to install GIMP by using the command line. Open a Terminal and enter the following command to install (this also works on Ubuntu):

```
sudo apt install gimp
```

You should also create a folder for your game. To do so, right-click your desktop anywhere there is empty space and select **New** ▸ **Folder** (on Ubuntu, the option is **Create New Folder** ; on macOS, it's **New Folder**). Enter *stickman* as the folder name.

CREATING THE GAME ELEMENTS

Once you've installed your graphics program, you're ready to draw. We'll create these images for our game elements:

- Images for a stick figure that can run left and right and jump
- Images for the platform, in three different sizes
- Images for the door: one open and one closed
- An image for the game's background (because a plain white or gray background makes for a boring game)

Before we start drawing, we need to prepare our images with transparent backgrounds.

PREPARING A TRANSPARENT IMAGE

To set up an image with transparency, start GIMP and follow these steps:

1. Select **File** ▸ **New** .
2. In the dialog, enter 27 pixels for the image width and 30 pixels for its height.
3. Select **Layer** ▸ **Transparency** ▸ **Add Alpha Channel** .
4. Select **Select** ▸ **All** .
5. Select **Edit** ▸ **Cut** .

The end result should be an image filled with a checkerboard of dark gray and light gray, as shown in Figure 13-3 (zoomed in).

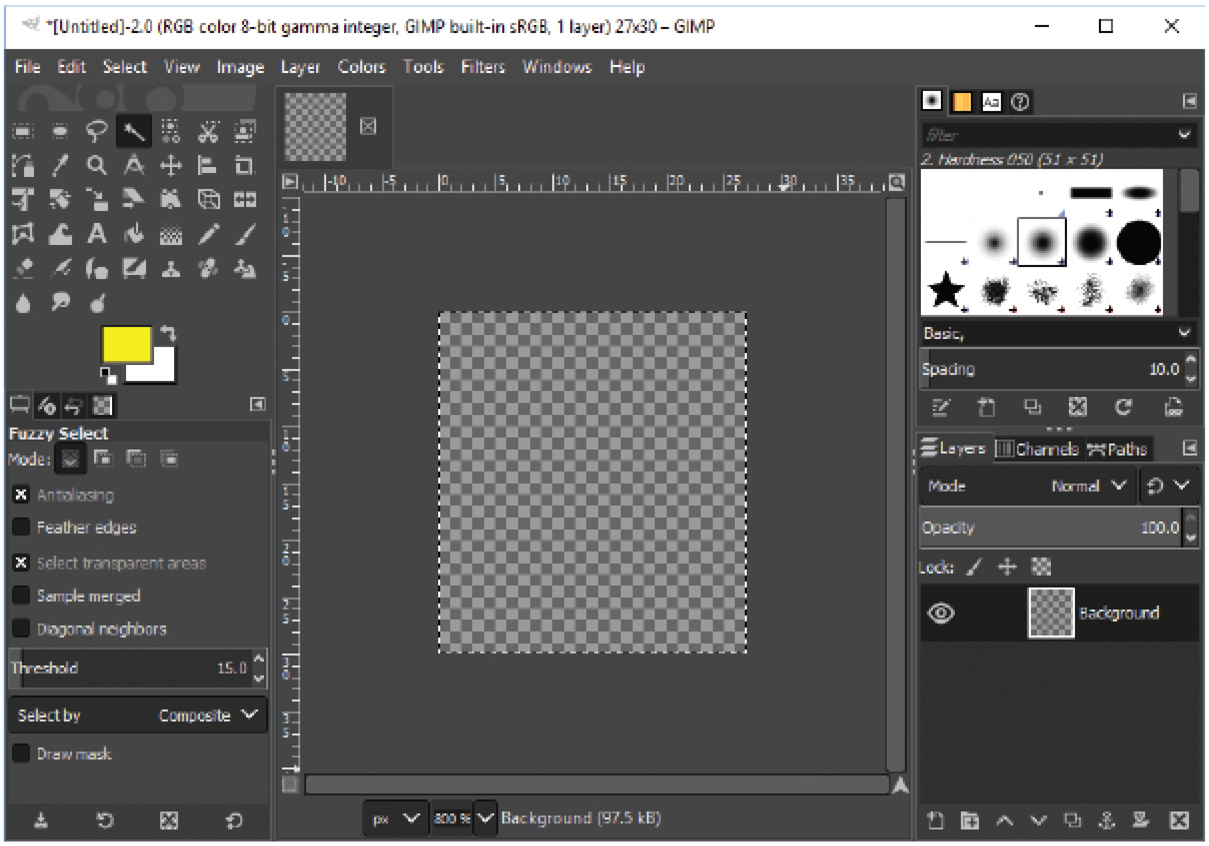


Figure 13-3: Zooming in on the transparent background

Now we can begin creating our secret agent: Mr. Stick Man.

DRAWING MR. STICK MAN

To draw our first stick figure image, click the Paintbrush tool in the GIMP Toolbox, and then select the brush that looks like a small dot in the Brushes toolbar (called *Pixel*), as shown in Figure 13-4 .

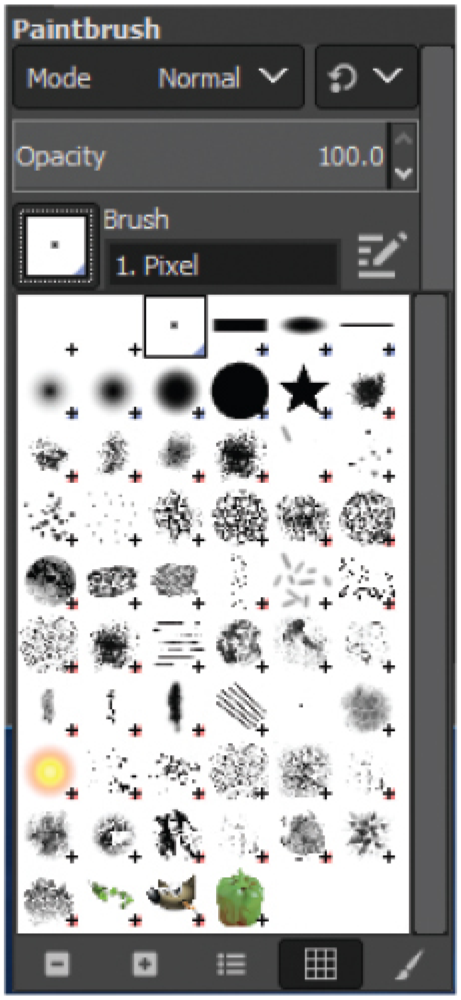


Figure 13-4: GIMP Toolbox

We'll draw three different images (or *frames*) for our stick figure to show him running and jumping to the right. We'll use these frames to animate Mr. Stick Man, as we did for the animation in Chapter 10 .

If you zoom in to look at these images, they might look like Figure 13-5 .

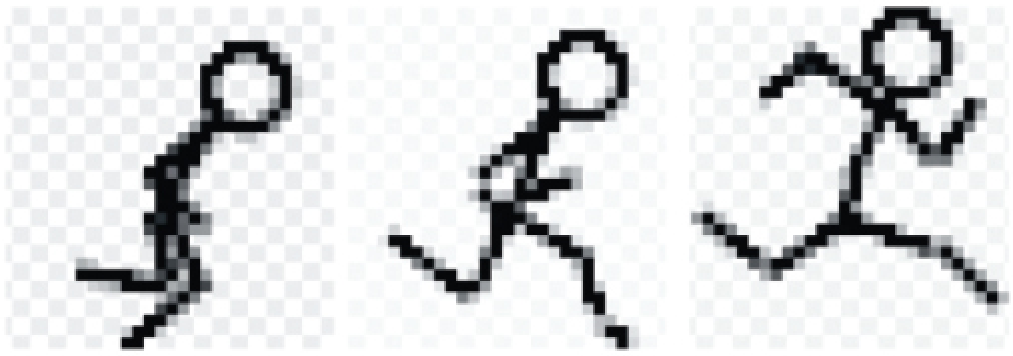


Figure 13-5: Zooming in on the stick figure

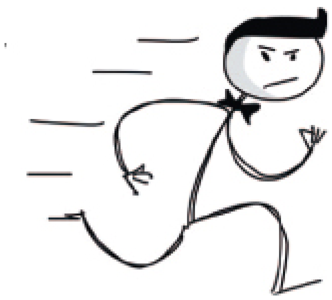
Your images don't need to look exactly the same, but they should have the stick figure in three different positions of movement. Each image should be 27 pixels wide by 30 pixels tall.

MR. STICK MAN RUNNING TO THE RIGHT

First, we'll draw a sequence of frames for Mr. Stick Man running to the right. Create the first image as follows:

1. Draw the first image (the leftmost image shown in Figure 13-5).
2. Select **File** ‣ **Save As** .
3. In the dialog, enter *figure-R1.gif* for the name. Then click the small plus (+) button labeled **Select File Type** .
4. Select **GIF image** in the list that appears.
5. Save the file to the *stickman* folder you created earlier (click **Browse for Other Folders** to find the correct folder).

Follow the same steps to create a new 27 by 30 pixel image for the next Mr. Stick Man in Figure 13-5 . Save this image as *figure-R2.gif* . Repeat the process for the final image, and save it as *figure-R3.gif* .



MR. STICK MAN RUNNING TO THE LEFT

Rather than re-creating our drawings for the stick figure moving to the left, we can use GIMP to flip our frames of Mr. Stick Man moving to the right.

In GIMP, open each image in sequence, and then select **Tools ▸ Transform Tools ▸ Flip** . When you click the image, you should see it flip from side to side, as in Figure 13-6 . Save the images as *figure-L1.gif* , *figure-L2.gif* , and *figure-L3.gif* .

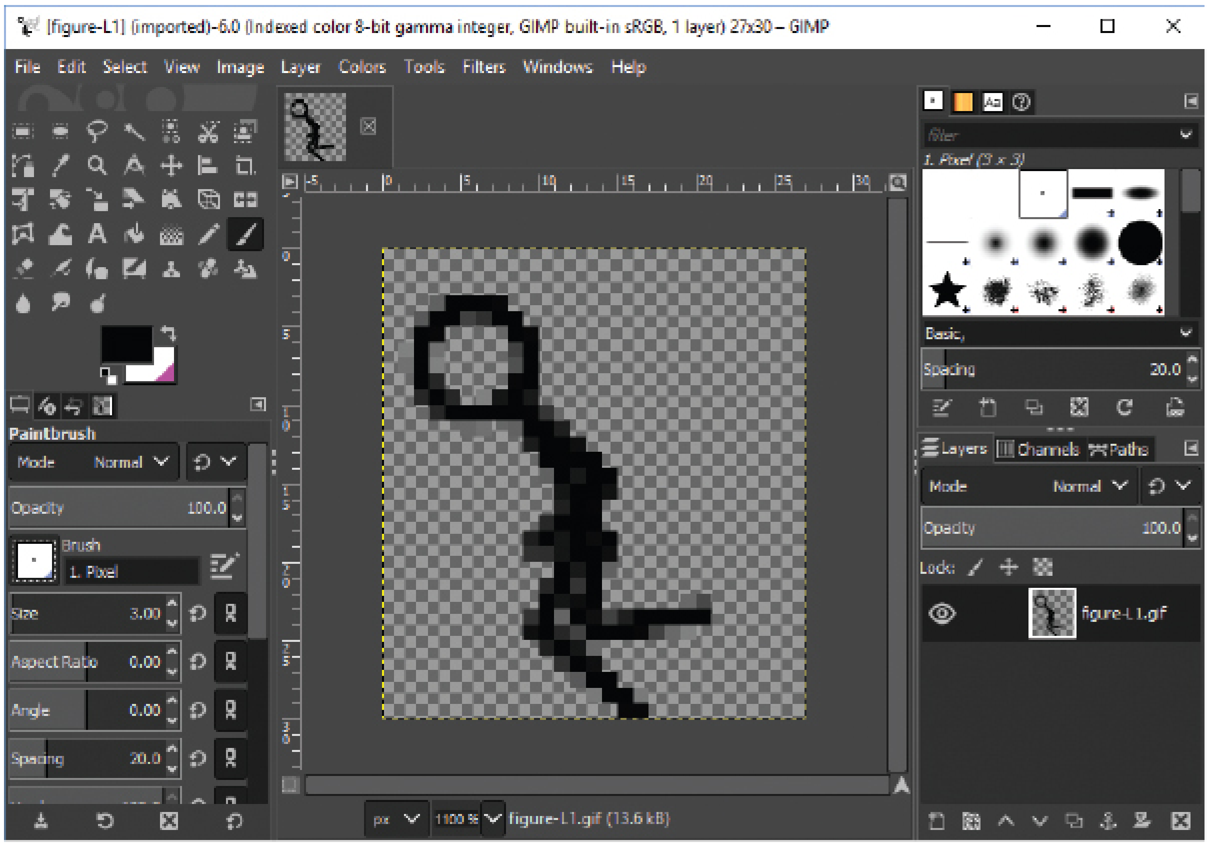

[figure-L1] (imported)-6.0 (Indexed color 8-bit gamma integer, GIMP built-in sRGB, 1 layer) 27x30 – GIMP
File Edit Select View Image Layer Colors Tools Filters Windows Help
Paintbrush
Mode Normal
Opacity 100.0
Brush 1. Pixel
Size 3.00
Aspect Ratio 0.00
Angle 0.00
Spacing 20.0
1. Pixel (3 × 3)
Basic,
Spacing 20.0
Layers Channels Paths
Mode Normal
Opacity 100.0
Lock:
figure-L1.gif
px 1100 % figure-L1.gif (13.6 kB)

Figure 13-6: Flipped stick figure

Now we've created six images for Mr. Stick Man, but we still need images for the platforms, the door, and the background.

DRAWING THE PLATFORMS

We'll create three platforms in different sizes: 100 pixels wide by 10 pixels tall; 66 pixels wide by 10 pixels tall; and 32 pixels wide by 10 pixels tall. You can draw the platforms any way you like, but make sure their backgrounds are transparent, as with the stick figure images.

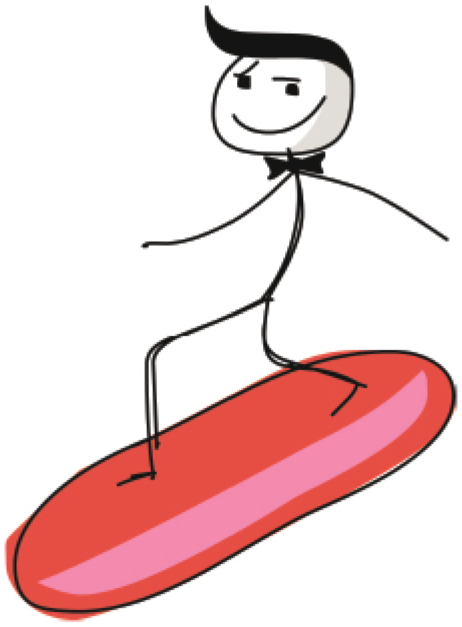


Figure 13-7 shows what the platforms might look like when we zoom in.

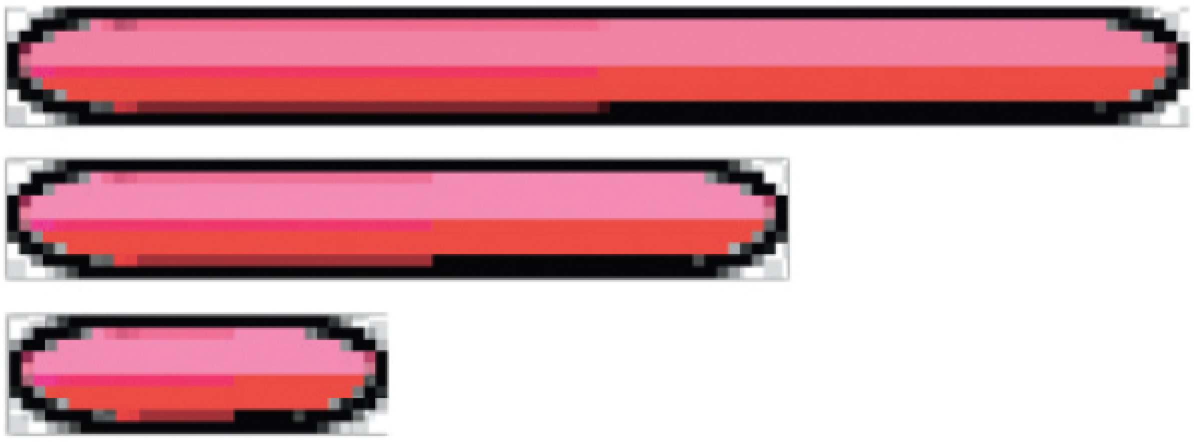


Figure 13-7: Zooming in on the platforms

As with the stick figure images, save these in the *stickman* folder. Call the largest platform *platform1.gif*, the middle-sized one *platform2.gif*, and the smallest *platform3.gif*.

DRAWING THE DOOR

The size of the door should be proportional to the size of Mr. Stick Man (27 pixels wide by 30 pixels tall). We'll need two images: one for the closed door and another for the open door. The doors might look like Figure 13-8 (zoomed in).

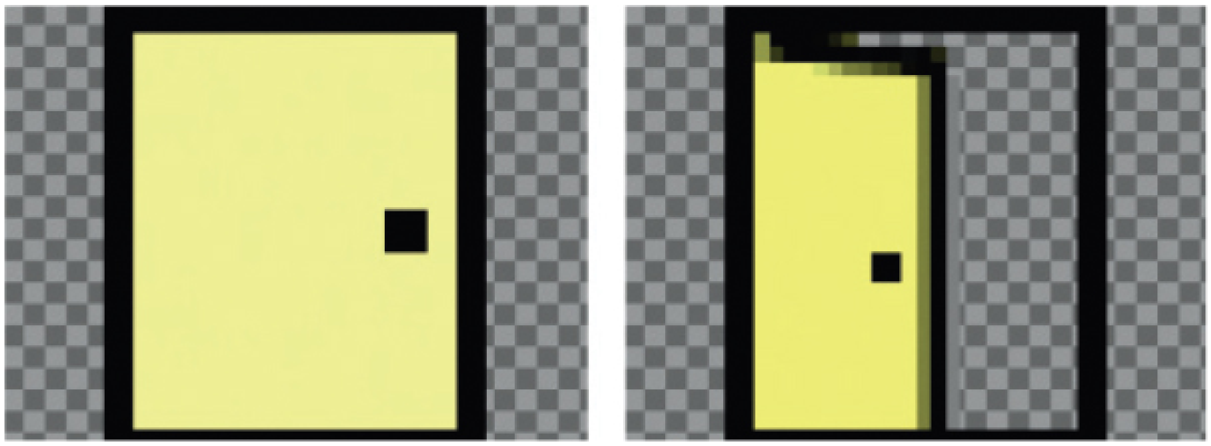


Figure 13-8: Zooming in on the doors

To create these images, follow these steps:

1. Click the foreground color box (at the bottom of the GIMP Toolbox) to display the color picker.
2. Select the color you want for your door. Figure 13-9 shows an example with yellow selected.
3. Choose the Bucket tool (shown selected in the Toolbox), and fill the screen with the color you chose.

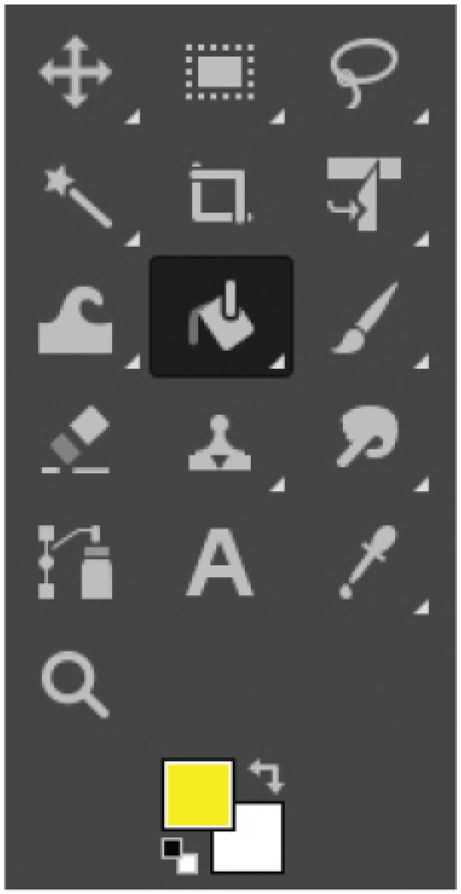


Figure 13-9: GIMP showing the background color selector

4. Change the foreground color to black.
5. Choose either the Pencil or Paintbrush tool (both are found to the right of the Bucket tool), and draw the black outline of the door and the doorknob.
6. Save these in the *stickman* folder and call them *door1.gif* and *door2.gif* .

DRAWING THE BACKGROUND

The final image we need to create is the background. We'll make this image 100 pixels wide by 100 pixels tall. It does not need a transparent background because we'll fill it with a single color that will be the background "wallpaper" behind all the other elements of the game.

To create the background, select **File ▸ New** and enter the image size as 100 pixels wide and 100 pixels tall. Choose a suitably evil color for the wallpaper of a villain's lair. I chose a darker shade of pink.

You can dress up your wallpaper with flowers, stripes, stars—whatever you like for your game. For example, to add stars to the wallpaper, choose another color, select the Pencil tool, and draw your first star. Then use the Selection tool to select a box around the star, and copy and paste it around the image (select **Edit ▸ Copy** , and then **Edit ▸ Paste**). You should be able to drag the pasted image around the screen by clicking it. Figure 13-10 shows an example with some stars, and the Selection tool selected in the Toolbox.

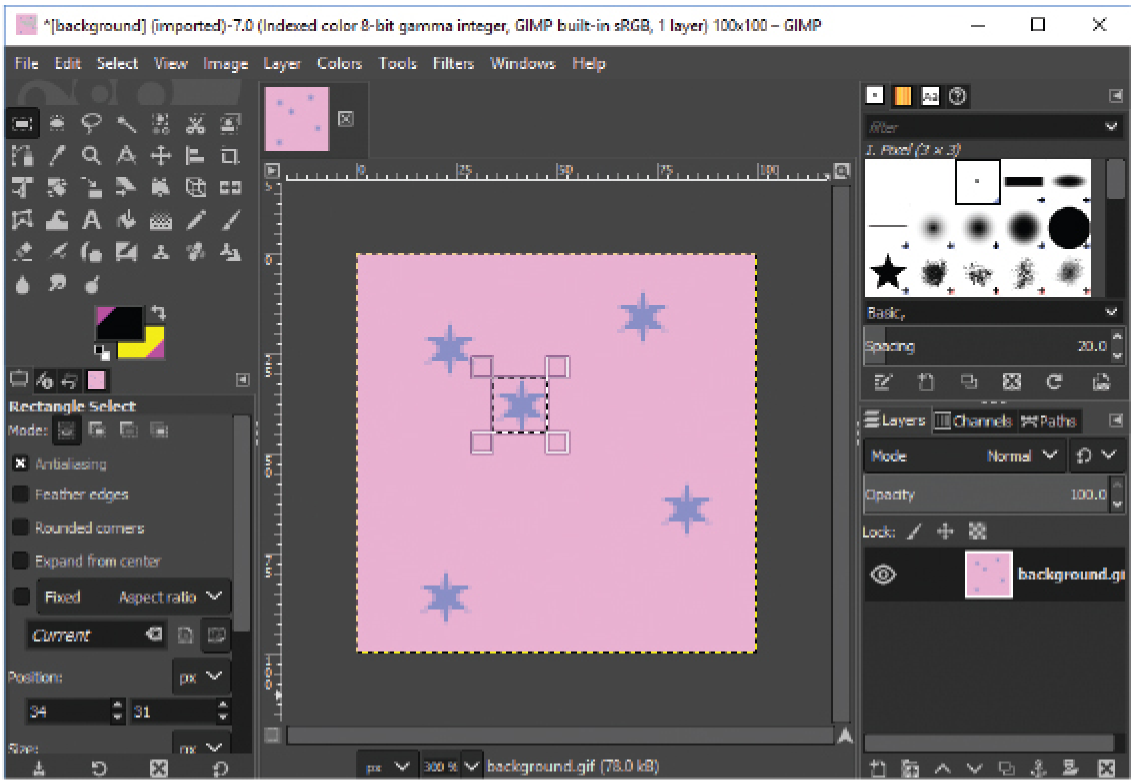


Figure 13-10: GIMP selection tool

Once you're happy with your drawing, save the image as *background.gif* in the *stickman* folder.

TRANSPARENCY

With our graphics created, you can get a better idea of why our images (other than the background) need transparency. If we placed Mr. Stick Man in front of our background wallpaper and he didn't have a transparent background, our game would look like Figure 13-11 .

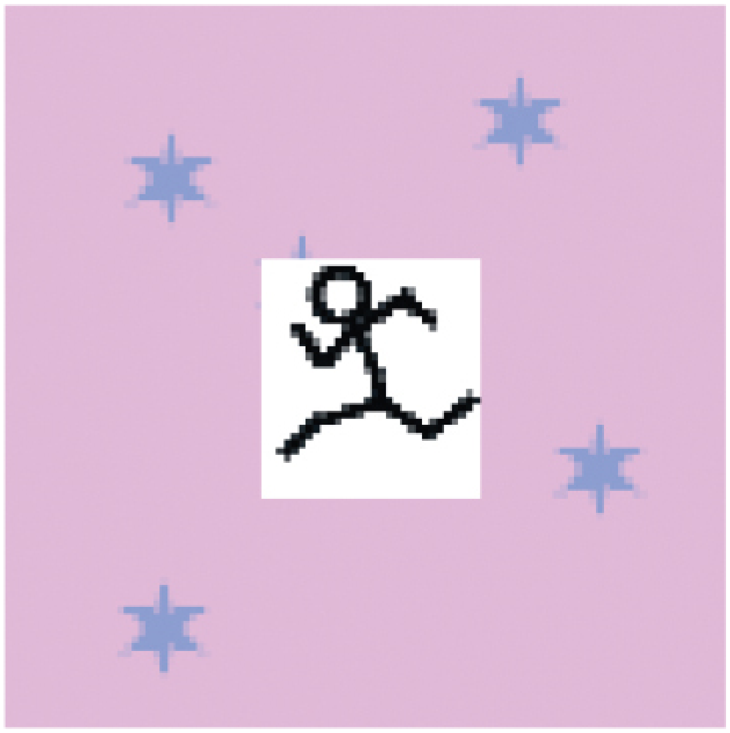


Figure 13-11: Stick figure with no transparency

The white background of Mr. Stick Man wipes out part of the wallpaper. If we use our transparent image, we get Figure 13-12 .

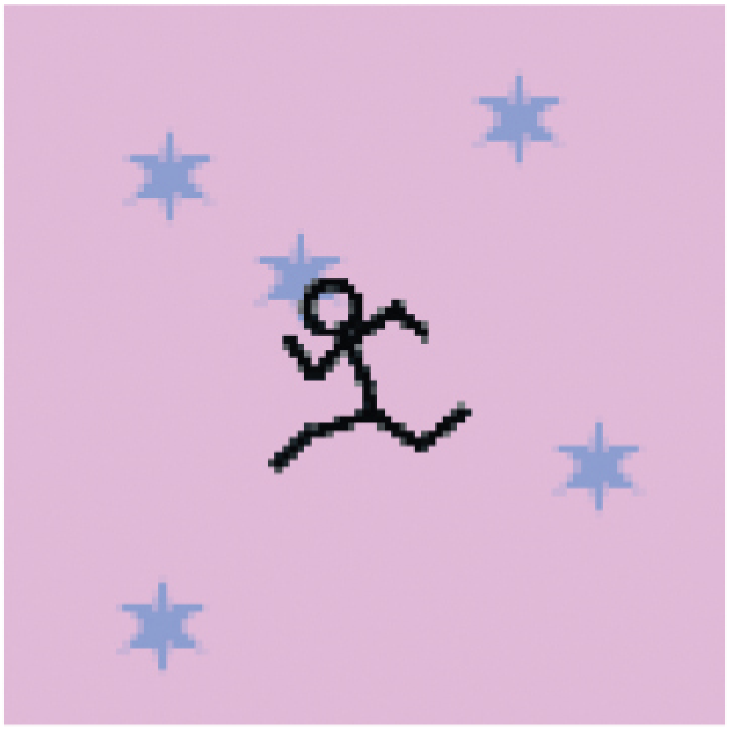
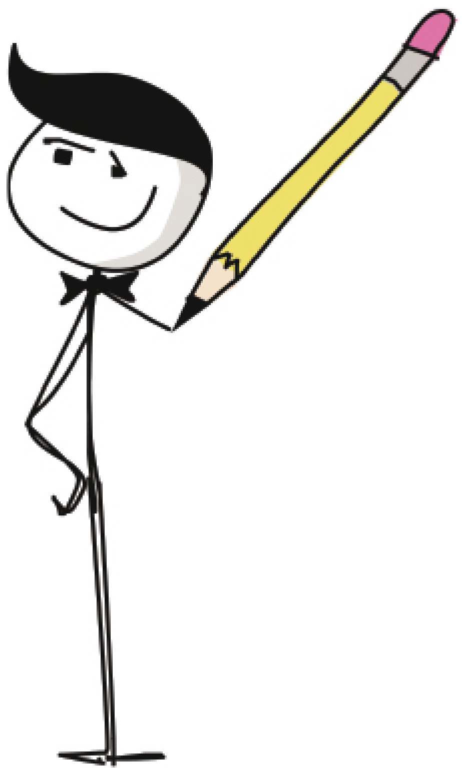


Figure 13-12: Stick figure with transparency

Nothing in the background is obscured by the stick figure image, except for whatever he covers himself. That's much more professional!

WHAT YOU LEARNED

In this chapter, you learned how to write a basic plan for a game and figured out where to begin. Because we need graphical elements before we can make a game, we used a graphics program to create basic frames. In the process, you learned how to make the backgrounds of these images transparent so they don't cover up other images on the screen.



In the next chapter, we'll create some of the classes for our game.

14
DEVELOPING THE MR. STICK MAN GAME

Now that we've created the images for *Mr. Stick Man Races for the Exit* , we can begin developing the code. The description of the game in the previous chapter gives us an idea what we'll need: a stick figure that can run and jump and platforms he must jump to. We'll write code to display the stick figure and move it across the screen, as well as to display the platforms. But before we write this code, we need to create the canvas to display our background image.

CREATING THE GAME CLASS

First, we'll create a class called `Game` that will be our program's main controller. The `Game` class will have an `__init__` function for initializing the game and a `mainloop` function for doing the animation.

SETTING THE WINDOW TITLE AND CREATING THE CANVAS

In the first part of the `__init__` function, we'll set the window title and create the canvas. As you'll see, this part of the code is similar to the code we wrote for *Bounce!* in Chapter 11 . Open a new file in IDLE and enter the following code, and then save your file as *stickmangame.py* . Make sure you save it in the *stickman* folder we created in Chapter 13 :

```
from tkinter import *
import random
import time

class Game:
    def __init__(self):
        self.tk = Tk()
        self.tk.title('Mr. Stick Man Races for the Exit')
        self.tk.resizable(0, 0)
        self.tk.wm_attributes('-topmost', 1)
        self.canvas = Canvas(self.tk, width=500, height=500,
                             highlightthickness=0)
        self.canvas.pack()
        self.tk.update()
        self.canvas_height = self.canvas.winfo_height()
        self.canvas_width = self.canvas.winfo_width()
```

In the first half of this program (the lines from `tkinter import *` to `self.tk.wm_attributes`), we create the `tk` object and then set the window title with `self.tk.title` to ("Mr. Stick Man Races for the Exit"). We make the window *fixed* (so it can't be resized) by calling the `resizable` function, and then we move the window in front of all other windows with the `wm_attributes` function.

Next, we create the canvas with the `self.canvas = Canvas` line, and call the `pack` and `update` functions of the `tk` object. Finally, we create two variables for our `Game` class, `height` and `width` , to store the height and width (we use the `winfo_height` and `winfo_width` functions to get the size of the canvas).

Note

The backslash (\) in the line `self.canvas = Canvas` is used only to separate a long line of code. It's not required in this case, but I've included it here for readability as the entire line won't fit on the page.

FINISHING THE __INIT__ FUNCTION

Now enter the rest of the `__init__` function into the *stickman game.py* file you just created. This code will load the background image and then display it on the canvas:

```
        self.tk.update()
        self.canvas_height = self.canvas.winfo_height()
        self.canvas_width = self.canvas.winfo_width()
        self.bg = PhotoImage(file='background.gif')
        w = self.bg.width()
        h = self.bg.height()
❶       for x in range(0, 5):
❷           for y in range(0, 5):
                self.canvas.create_image(x * w, y * h,
                        image=self.bg, anchor='nw')
        self.sprites = []
        self.running = True
```

At the line beginning `self.bg`, we create the variable `bg`, which contains a `PhotoImage` object—the background image file called *background.gif* that we created in Chapter 13 on page 210 . Next, we store the width and height of the image in the `w` and `h` variables. The `PhotoImage` class functions `width` and `height` return the size of the image once it's been loaded.

Next come two loops inside this function. To understand what they do, imagine you have a small square rubber stamp, an ink pad, and a large piece of paper. How can you use the stamp to fill the paper with colored squares? Well, you could just randomly cover the page with stamps until it's filled. The result would be a mess, and it would take a while to complete, but it would fill the page. Or you could start stamping down in a column and then move back to the top and start stamping down the page in the next column, as shown in Figure 14-1 .

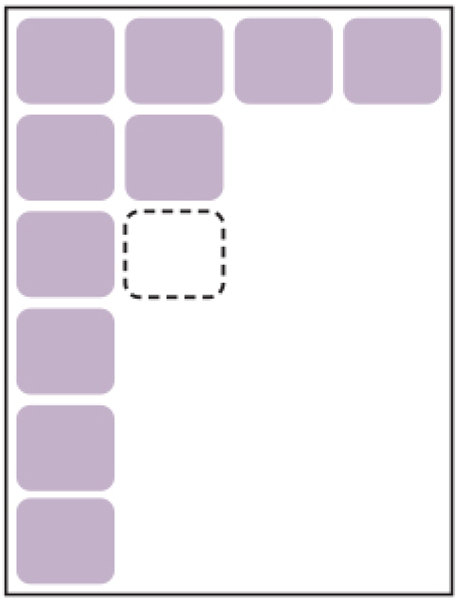


Figure 14-1: Stamping down the page

The background image we created in the previous chapter is our stamp. We know that the canvas is 500 pixels across and 500 pixels down, and that we created a background image of 100 pixels square. This tells us that we need five columns across and five rows down to fill the screen with images. We use a `for` loop ❶ to calculate the columns across, and another `for` loop ❷ to calculate rows going down.

After this, we multiply the first loop variable `x` by the width of the image (`x * w`) to determine how far across we're drawing, and then multiply the second loop variable `y` by the height of the image (`y * h`) to calculate how far down to draw. We use the `create_image` function of the canvas object (`self.canvas.create _image`) to draw the image on the screen using those coordinates.

Finally, we create the variables `sprites` , which holds an empty list, and `running` , which contains the `True` Boolean value. We'll use these variables later in our game code.

CREATING THE MAINLOOP FUNCTION

We'll use the `mainloop` function in the `Game` class to animate our game. This function looks a lot like the main loop (or animation loop) we created for the *Bounce!* game in Chapter 11 .

Our function is as follows:

```
        for x in range(0, 5):
            for y in range(0, 5):
                self.canvas.create_image(x * w, y * h,
                        image=self.bg, anchor='nw')
        self.sprites = []
        self.running = True

    def mainloop(self):
        while True:
           if self.running == True:
               for sprite in self.sprites:
                   sprite.move()
            self.tk.update_idletasks()
            self.tk.update()
            time.sleep(0.01)
```

We create a `while` loop that will run until the game window is closed (`while True` is an infinite loop, which we first saw back on page 175). Next, we check to see if the `running` variable is equal to `True` . If it is, we loop through any sprites in the list of sprites (`self.sprites`), calling the `move` function for each one. (We have yet to create any sprites, so this code won't do anything if we run the program now, but it will be useful later.)

The last three lines of the function force the `tk` object to redraw the screen and sleep for a fraction of a second, as we did with the *Bounce!* game.

So you can run this code, add the following two lines (note that there's no indentation) and save the file:

```
g = Game()
g.mainloop()
```

Note

Be sure to add this code to the bottom of your game file. Also, make sure that your images and Python file are all in the stickman *folder you created in Chapter 13 .*

This code creates an object of the `Game` class and saves it as the `g` variable. We then call the `mainloop` function on the new object to draw the screen.

Once you've saved the program, run it in IDLE with **Run ‣ Run Module** . A window should appear with the background image filling the canvas, as in Figure 14-2 .

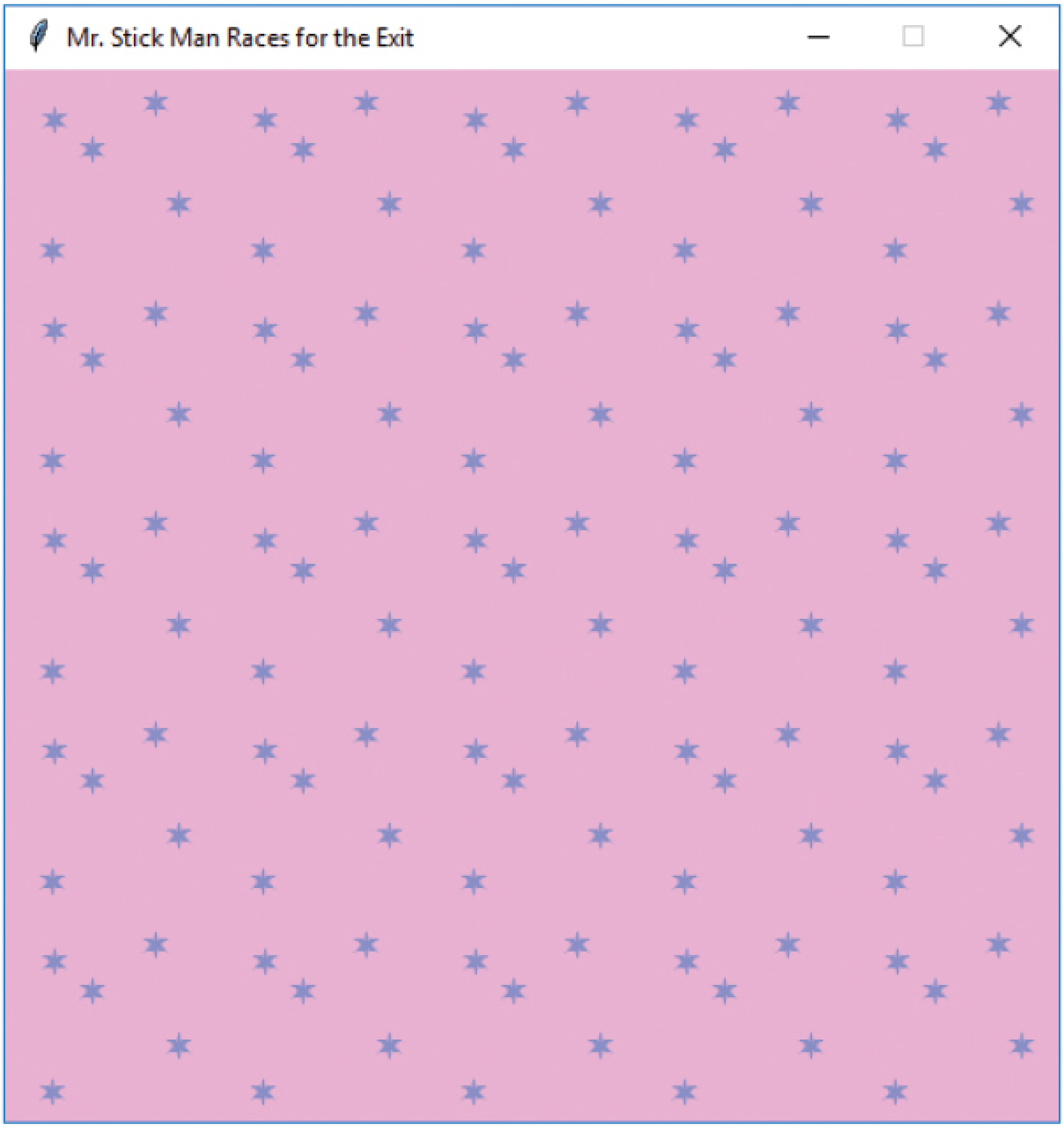


Figure 14-2: Game background

With this, we've added a nice background for our game and created an animation loop that will draw sprites for us (once we've created them).

CREATING THE COORDS CLASS

Now we'll create the class that we'll use to specify the position of sprites on our game screen. This class will store the top-left (*x1* and *y1*) and bottom-right (*x2* and *y2*) coordinates of any component of our game.

Figure 14-3 shows how you might record the position of the stick figure image using these coordinates.

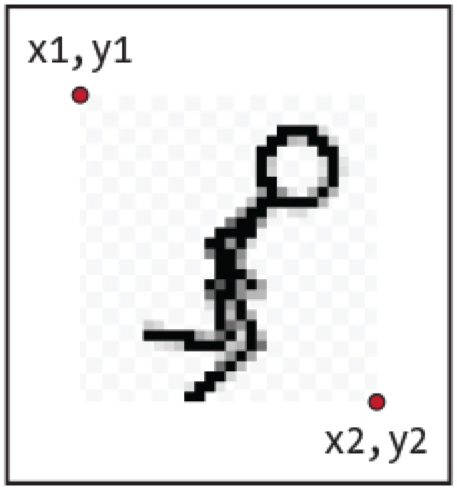


Figure 14-3: Where the x and y coordinates can be found on the stick figure

Our new class, `Coords` , will contain only an `__init__` function, to which we pass the four parameters (`x1` , `y1` , `x2` , and `y2`). Put this code at the beginning of the *stickmangame.py* file:

```
class Coords:
    def __init__(self, x1=0, y1=0, x2=0, y2=0):
        self.x1 = x1
        self.y1 = y1
        self.x2 = x2
        self.y2 = y2
```

Notice that each parameter is saved as an object variable of the same name (`x1` , `y1` , `x2` , and `y2`). We'll be using objects of this class shortly.

CHECKING FOR COLLISIONS

Once we know how to store the position of our game sprites, we need a way to tell if one sprite has collided with another, like when Mr. Stick

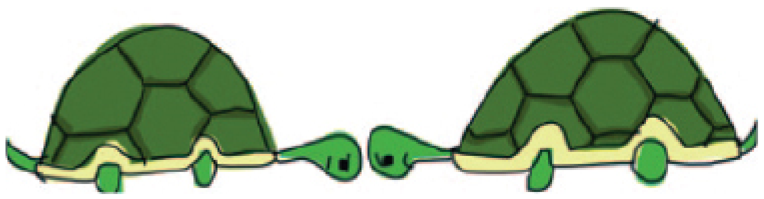
Man jumps around the screen and runs into one of the platforms. To make this problem easier to solve, we can break it into two smaller problems: checking if sprites are colliding vertically, and checking if sprites are colliding horizontally. We can then combine our solutions to see if two sprites are colliding in any direction!

SPRITES COLLIDING HORIZONTALLY

First, we'll create the `within_x` function to determine if one set of *x* coordinates (*x1* and *x2*) has crossed over another set of *x* coordinates (again, *x1* and *x2*). Add the following directly below the `Coords` class:

```
class Coords:
    def __init__(self, x1=0, y1=0, x2=0, y2=0):
        self.x1 = x1
        self.y1 = y1
        self.x2 = x2
        self.y2 = y2

def within_x(co1, co2):
 ❶ if co1.x1 > co2.x1 and co1.x1 < co2.x2:
        return True
 ❷ elif co1.x2 > co2.x1 and co1.x2 < co2.x2:
        return True
    elif co2.x1 > co1.x1 and co2.x1 < co1.x2:
        return True
    elif co2.x2 > co1.x1 and co2.x2 < co1.x2:
        return True
    else:
        return False
```

The `within_x` function takes the parameters `co1` and `co2` , both `Coords` objects. We first check to see if the leftmost position of the first coordinate object (`co1.x1`) is between the leftmost position (`co2.x1`) and the rightmost position (`co2.x2`) of the second coordinate object ❶ . We return `True` if it is.



Let's take a look at two lines with overlapping *x* coordinates to understand how this works. Each line in Figure 14-4 starts at `x1` and finishes at `x2` .

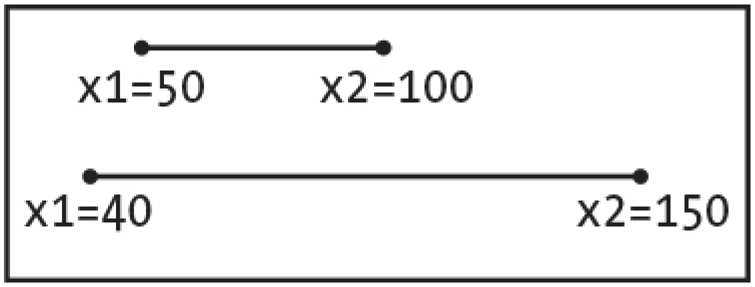


Figure 14-4: Overlapping horizontal (x) coordinates

The first line in this diagram (`co1`) starts at pixel position 50 (`x1`) and finishes at 100 (`x2`). The second line (`co2`) starts at position 40 and finishes at 150. In this case, because the `x1` position of the first line is between the `x1` and `x2` positions of the second line, the `if` statement in the function would be true for these two sets of coordinates.

With the first `elif` statement ❷ , we see whether the rightmost position of the first line (`co1.x2`) is between the leftmost position (`co2.x1`) and rightmost position (`co2.x2`) of the second. If it is, we return `True` . The next two `elif` statements do almost the same thing: they check the leftmost and rightmost positions of the second line (`co2`) against the first (`co1`).

If none of the `if` statements match, we reach `else` and return `False` . This is effectively saying, "No, the two coordinate objects do not cross over each other horizontally."

To see an example of the function working, look back at Figure 14-4 . The `x1` and `x2` positions of the first coordinate object are 50 and 100, and the `x1` and `x2` positions of the second coordinate object are 40 and

150. Here's what happens when we call the `within_x` function we've created:

```
>>> c1 = Coords(50, 50, 100, 100)
>>> c2 = Coords(40, 40, 150, 150)
>>> print(within_x(c1, c2))
True
```

The function returns `True`. This is the first step in determining whether one sprite has bumped into another. For example, when we create a class for Mr. Stick Man and for the platforms, we'll be able to tell if their *x* coordinates have crossed one another.

It's not best practice to have lots of `if` or `elif` statements that return the same value. To solve this problem, we can shorten the `within_x` function by surrounding each of its conditions with parentheses, separated by the `or` keyword. For a slightly neater function with fewer lines of code, you can change the function so it looks like this:

```
def within_x(co1, co2):
    if (co1.x1 > co2.x1 and co1.x1 < co2.x2) \
            or (co1.x2 > co2.x1 and co1.x2 < co2.x2) \
            or (co2.x1 > co1.x1 and co2.x1 < co1.x2) \
            or (co2.x2 > co1.x1 and co2.x2 < co1.x2):
        return True
    else:
        return False
```

To extend the `if` statement across multiple lines so that we don't end up with one really long line containing all the conditions, we use a backslash (\), as shown above.

SPRITES COLLIDING VERTICALLY

We also need to know if sprites collide vertically. The `within_y` function is very similar to the `within_x` function. To create it, we check whether the *y1* position of the first coordinate has crossed over the *y1* and *y2* positions of the second, and then vice versa.

Add the following function below the `within_x` function. This time, we'll use the shorter version of the code (rather than lots of `if` statements):

```
def within_y(co1, co2):
    if (co1.y1 > co2.y1 and co1.y1 < co2.y2) \
            or (co1.y2 > co2.y1 and co1.y2 < co2.y2) \
            or (co2.y1 > co1.y1 and co2.y1 < co1.y2) \
            or (co2.y2 > co1.y1 and co2.y2 < co1.y2):
        return True
    else:
        return False
```

Our `within_x` and `within_y` functions look quite similar because, in the end, they are doing similar things.

PUTTING IT ALL TOGETHER: OUR FINAL COLLISION-DETECTION CODE

Once we've determined whether one set of *x* coordinates has crossed over another, and done the same for *y* coordinates, we can write functions to see whether a sprite has hit another sprite and on which side. We'll do this with the `collided_left` , `collided_right` , `collided_top` , and `collided_bottom` functions.

THE COLLIDED_LEFT FUNCTION

Add the following code for the `collided_left` function below the two `within` functions we just created:

```
def collided_left(co1, co2):
    if within_y(co1, co2):
        if co1.x1 >= co2.x1 and co1.x1 <= co2.x2:
            return True
    return False
```

This function tells us whether the left-hand side (the `x1` value) of a first coordinate object has hit another coordinate object.

The function takes two parameters: `co1` (the first coordinate object) and `co2` (the second coordinate object). We check whether the two coordinate objects have crossed over vertically, using the `within_y` function. After all, there's no point in checking whether Mr. Stick Man has hit a platform if he is floating way above it (like Figure 14-5).

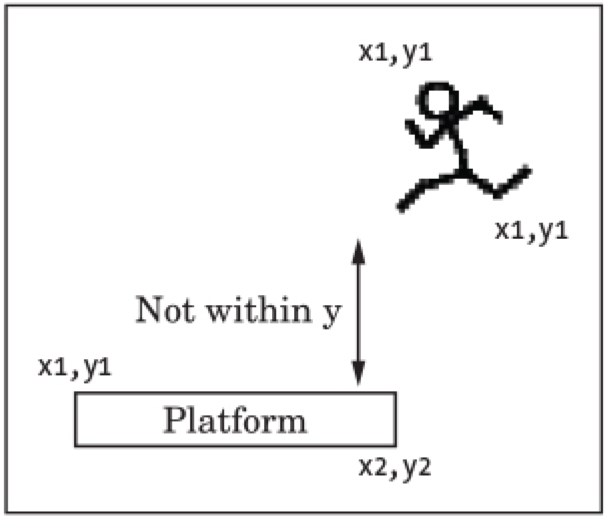
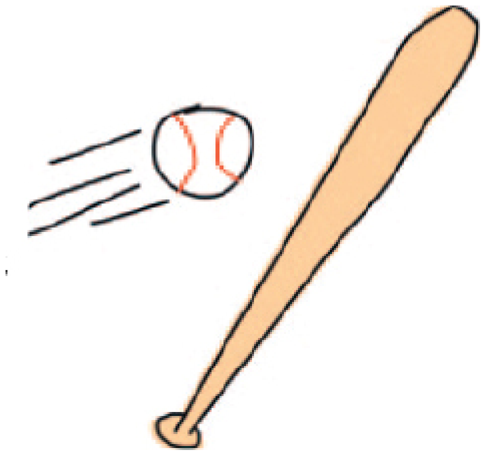


Figure 14-5: Mr. Stick Man above the platform

Then, we see if the value of the leftmost position of the first coordinate object (`co1.x1`) has hit the `x2` position of the second coordinate object (`co2.x2`). If so, it should be less than or equal to the `x2` position. We also check to make sure that it hasn't gone past the `x1` position. If it has hit the side, we return `True` . If none of the `if` statements are true, we return `False` .



THE COLLIDED_RIGHT FUNCTION

The `collided_right` function looks a lot like `collided_left` :

```
def collided_right(co1, co2):
    if within_y(co1, co2):
        if co1.x2 >= co2.x1 and co1.x2 <= co2.x2:
            return True
    return False
```

As with `collided_left` , we check to see if the *y* coordinates have crossed over each other, using the `within_y` function. We then check to see if the `x2` value of the first coordinate object is between the `x1` and `x2` positions of the second coordinate object, and return `True` if it is. Otherwise, we return `False` .

THE COLLIDED_TOP FUNCTION

The `collided_top` function is very similar to the two functions we just added:

```
def collided_top(co1, co2):
    if within_x(co1, co2):
        if co1.y1 >= co2.y1 and co1.y1 <= co2.y2:
            return True
    return False
```

This time, we check to see if the coordinates have crossed over horizontally, using the `within_x` function. Next, we see if the topmost position of the first coordinate (`co1.y1`) has crossed over the `y2` position of the second coordinate, but not its `y1` position. If so, we return `True` (the top of the first coordinate has hit the second coordinate).

THE COLLIDED_BOTTOM FUNCTION

Our last function, `collided_bottom` , is just a bit different:

```
def collided_bottom(y, co1, co2):
    if within_x(co1, co2):
        y_calc = co1.y2 + y
❶     if y_calc >= co2.y1 and y_calc <= co2.y2:
            return True
    return False
```

This function takes an additional parameter, `y` , a value that we add to the `y` position of the first coordinate. Our `if` statement checks if the coordinates have crossed over horizontally (as we did with `collided_top`). Next, we add the value of the `y` parameter to the first coordinate's `y2` position, and store the result in the `y_calc` variable. If the newly calculated value is between the `y1` and `y2` values of the second coordinate ❶ , we return `True` because the bottom of coordinate `co1` has hit the top of coordinate `co2` . However, if none of the `if` statements are true, we return `False` .

We need the additional `y` parameter because Mr. Stick Man could fall off a platform. Unlike the other `collided` functions, we need to be able to test if he *would* collide at the bottom, rather than whether he already has. If he walks off a platform and keeps floating in midair, our game won't be very realistic; so as he walks, we check to see if he has collided with something on the left or right. When we check below him, we see if he would collide with the platform; if not, he needs to go crashing down!

CREATING THE SPRITE CLASS

The parent class for our game items, `Sprite` , will provide two functions: `move` to move the sprite, and `coords` to return the sprite's current position on the screen. We add the code for the `Sprite` class below the `collided_bottom` function, as follows:

```
class Sprite:
    def __init__(self, game):
        self.game = game
        self.endgame = False
        self.coordinates = None

    def move(self):
        pass

    def coords(self):
        return self.coordinates
```

The Sprite class's `__init__` function takes a single parameter, `game` , which will be the game object. We need it so that any sprite we create will be able to access the list of other sprites in the game. We store the game parameter as an object variable.

Then, we store the object variable `endgame` , which we'll use to indicate the end of the game. (At the moment, it's set to `False` .) The final object variable, `coordinates` , is set to nothing (`None`).

The `move` function does nothing in this parent class, so we use the `pass` keyword in the body of this function. The `coords` function simply returns the object variable `coordinates` .

So our `Sprite` class has a `move` function that does nothing and a `coords` function that returns no coordinates. That doesn't sound very useful, does it? However, any classes that have `Sprite` as their parent will always have the `move` and `coords` functions. So, in the main loop of the game, when we loop through a list of sprites, calling the `move` function won't cause any errors because each sprite has that function.

Note

Classes with functions that don't do very much are quite common in programming. In a way, they're an agreement that ensures all the children of a class provide the same sort of functionality, even if in some cases the functions in the child classes do nothing.

ADDING THE PLATFORMS

Now we'll add the platforms. Our class for platform objects, `PlatformSprite` , will be a child class of `Sprite` . The `__init__` function for this class will take a `game` parameter (as the `Sprite` parent class does), as well as an image, `x` and `y` positions, and the image `width` and `height` . Here's the code for the `PlatformSprite` class, which goes directly below the `Sprite` class:

```
class PlatformSprite(Sprite):
    def __init__(self, game, photo_image, x, y, width, height):
        Sprite.__init__(self, game)
        self.photo_image = photo_image
        self.image = game.canvas.create_image(x, y,
                image=self.photo_image, anchor='nw')
        self.coordinates = Coords(x, y, x + width, y + height)
```

When we define the `PlatformSprite` class, we give it a single parameter: the name of the parent class (`Sprite`). The `__init__` function has seven parameters: `self` , `game` , `photo_image` , `x` , `y` , `width` , and `height` .

We call the `__init__` function of the parent class, `Sprite` , using `self` and `game` as the parameter values, because other than the `self` parameter, the Sprite class's `__init__` function takes only one parameter: `game` .

At this point, if we were to create a `PlatformSprite` object, it would have all the object variables from its parent class (`game` , `endgame` , and `coordinates`), simply because we've called the `__init__` function in `Sprite` .

Next, we save the `photo_image` parameter as an object variable, and we use the `canvas` variable of the game object to draw the image onscreen with `create_image` .

Finally, we create a `Coords` object with the `x` and `y` parameters as the first two arguments. We then add the `width` and `height` parameters to these parameters for the second two arguments.

Even though the `coordinates` variable is set to `None` in the `Sprite` parent class, we've changed it in our `PlatformSprite` child class to an actual `Coords` object, containing the live location of the platform image on the screen.

ADDING A PLATFORM OBJECT

Let's add a platform to the game to see how it looks. Change the last two lines of the game file (*stickmangame.py*):

```
   g = Game()
❶ platform1 = PlatformSprite(g, PhotoImage(file='platform1.gif'),
                                   0, 480, 100, 10)
❷ g.sprites.append(platform1)
   g.mainloop()
```

We create an object of the `PlatformSprite` class, passing it the variable for our game (`g`), along with a `PhotoImage` object (which uses the first of our platform images, *platform1.gif*) ❶ . We also pass it the position where we want to draw the platform (0 pixels across and 480 pixels down, near the bottom of the canvas), along with the height and width of our image (100 pixels across and 10 pixels high). We add this sprite to the list of sprites in our game object ❷ .

If you run the game now, you should see a platform at the bottom-left side of the screen, like Figure 14-6 .

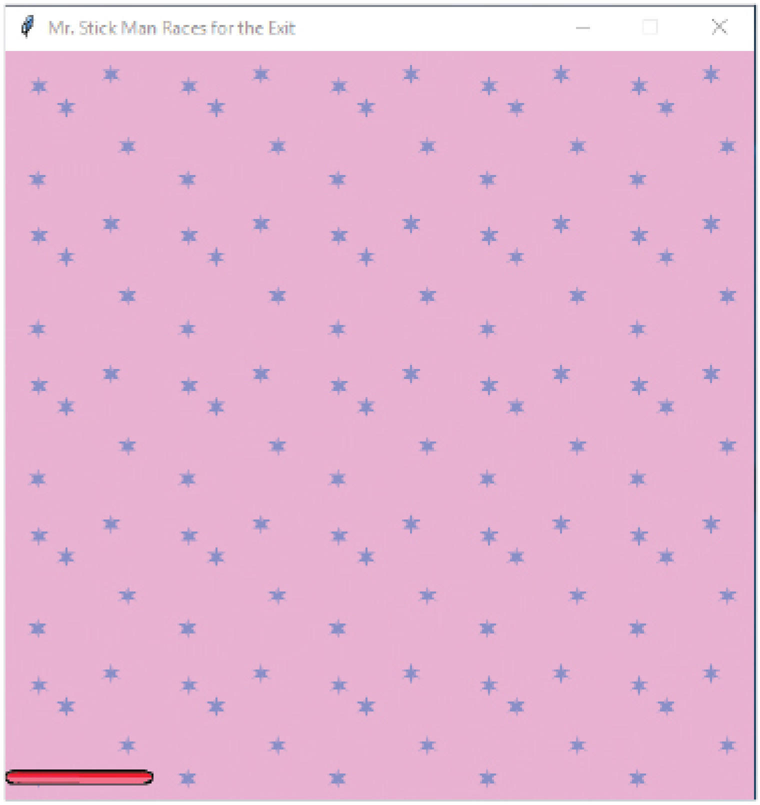


Figure 14-6: Displaying a platform

ADDING A BUNCH OF PLATFORMS

Let's add a whole bunch of platforms. Each platform will have different *x* and *y* positions, so they'll be scattered around the screen. Use the following code:

```
g = Game()
platform1 = PlatformSprite(g, PhotoImage(file='platform1.gif'),
                           0, 480, 100, 10)
platform2 = PlatformSprite(g, PhotoImage(file='platform1.gif'),
                           150, 440, 100, 10)
platform3 = PlatformSprite(g, PhotoImage(file='platform1.gif'),
                           300, 400, 100, 10)
platform4 = PlatformSprite(g, PhotoImage(file='platform1.gif'),
                           300, 160, 100, 10)
platform5 = PlatformSprite(g, PhotoImage(file='platform2.gif'),
                           175, 350, 66, 10)
platform6 = PlatformSprite(g, PhotoImage(file='platform2.gif'),
                           50, 300, 66, 10)
platform7 = PlatformSprite(g, PhotoImage(file='platform2.gif'),
                           170, 120, 66, 10)
platform8 = PlatformSprite(g, PhotoImage(file='platform2.gif'),
                           45, 60, 66, 10)
platform9 = PlatformSprite(g, PhotoImage(file='platform3.gif'),
                           170, 250, 32, 10)
platform10 = PlatformSprite(g, PhotoImage(file='platform3.gif'),
                            230, 200, 32, 10)
g.sprites.append(platform1)
g.sprites.append(platform2)
g.sprites.append(platform3)
g.sprites.append(platform4)
g.sprites.append(platform5)
g.sprites.append(platform6)
g.sprites.append(platform7)
g.sprites.append(platform8)
g.sprites.append(platform9)
g.sprites.append(platform10)
g.mainloop()
```

We first create a ton of `PlatformSprite` objects, saving them as variables `platform1` , `platform2` , `platform3` , and so on, up to `platform10` . We then add each platform to the `sprites` variable, which we created in our `Game` class. If you run the game now, it should look like Figure 14-7 .

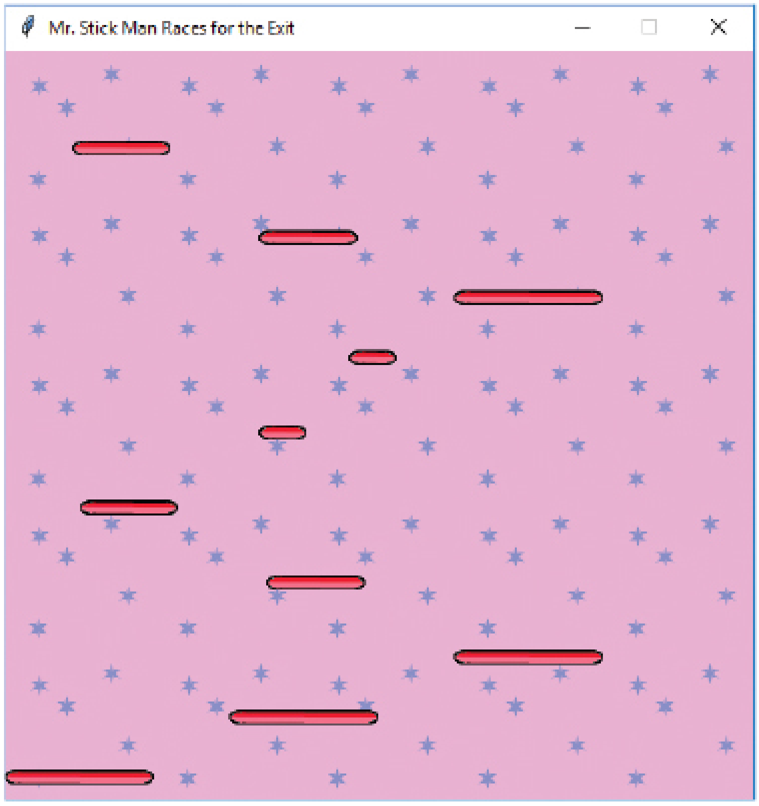


Figure 14-7: Displaying all the platforms

We've created the basics of our game! Now we're ready to add our main character, Mr. Stick Man.

WHAT YOU LEARNED

In this chapter, you created the `Game` class and drew the background image onto the screen. You learned how to determine whether a horizontal or vertical position is within the bounds of two other horizontal or vertical positions by creating the `within_x` and `within_y` functions. You then used these functions to create new functions that determine whether one coordinate object had collided with another. We'll use these functions in the next chapters when we animate Mr. Stick Man and need to detect whether he has collided with a platform as he moves around the canvas.

We also created a parent class `Sprite` and its first child class, `PlatformSprite` , which we used to draw the platforms onto the canvas.

PROGRAMMING PUZZLES

The following coding puzzles are some ways to experiment with the game's background image. Check your answers at *http://python-for-kids.com* .

#1: CHECKERBOARD

Try changing the `Game` class so that the background image is drawn like a checkerboard, as in Figure 14-8 .

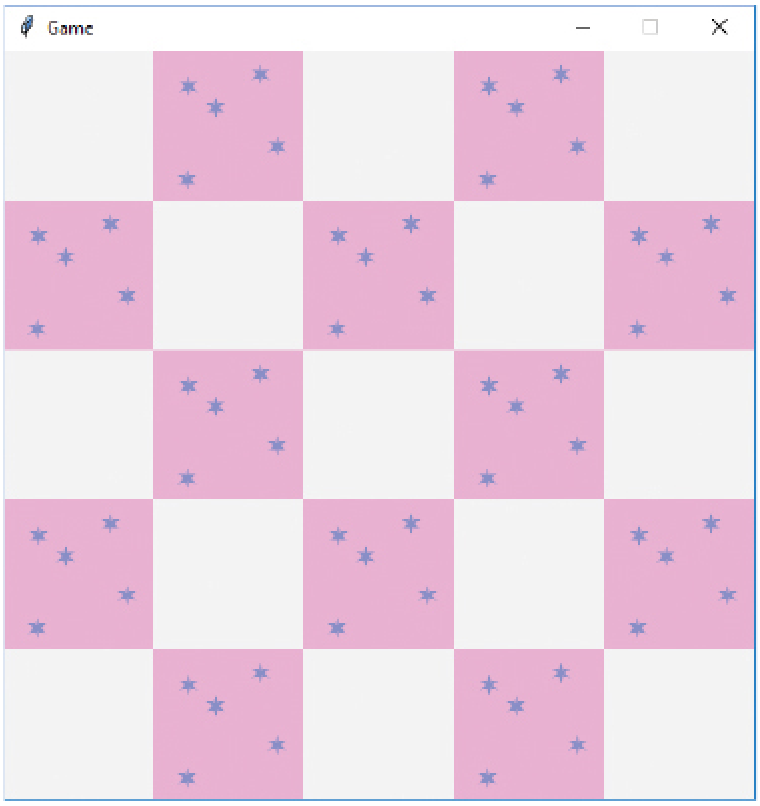


Figure 14-8: Background as a checkerboard

#2: TWO-IMAGE CHECKERBOARD

Once you've figured out how to create a checkerboard effect, try using two alternating images. Come up with another wallpaper image (using your graphics program), and then change the `Game` class so it displays a checkerboard with two alternating images instead of one image and the blank background.

#3: BOOKSHELF AND LAMP

You can create different wallpaper images to make the game's background more interesting. Create a copy of the background image; then draw a simple bookshelf, a table with a lamp, or a window. Dot these images around the screen by changing the `Game` class so that it displays a few different wallpaper images.

#4: RANDOM BACKGROUND

As an alternative to the two-image checkerboard, try creating five different background images. You can either draw them as a repeating pattern of background images (1, 2, 3, 4, 5, 1, 2, 3, 4, 5, and so on), or you can draw them randomly.

Hint: If you import the `random` module and put your images in a list, try using `random.choice()` to pick one randomly.

15
CREATING MR. STICK MAN

In this chapter, we'll create the main character of *Mr. Stick Man Races for the Exit* . This will require the most complicated coding we've done so far, because Mr. Stick Man needs to run left and right, jump, stop when he runs into a platform, and fall when he runs off the edge of a platform. We'll use event bindings for the left and right arrow keys to make the stick figure run left and right, and we'll have him jump when the player presses the spacebar.

INITIALIZING THE STICK FIGURE

The `__init__` function for our new stick figure class will look a lot like those in the other classes in our game. We start by giving our new class a name— `StickFigureSprite` —and assign this class to a parent class, `Sprite` :

```
class StickFigureSprite(Sprite):
    def __init__(self, game):
        Sprite.__init__(self, game)
```

This code looks similar to the `PlatformSprite` class in Chapter 14 , except we're not using any additional parameters (other than `self` and `game`). This is because, unlike the `PlatformSprite` class, only one `StickFigureSprite` object will be used in the game.

LOADING THE STICK FIGURE IMAGES

Because we have a lot of platform objects on the screen, we pass the platform image as a parameter of the `PlatformSprite` 's `__init__` function (kind of like saying, "Here, PlatformSprite, use this image when you draw yourself on the screen."). But since there's only one stick figure object, it doesn't make sense to load the image outside the sprite and then pass it in as a parameter. The `StickFigureSprite` class will know how to load its own images.

The next few lines of the `__init__` function load the three left images (which we'll use to animate the stick figure running left) and the three right images (used to animate the stick figure running right). We need to load these images now, because we don't want to load them every time we display the stick figure on the screen (doing so would take too long and make our game run slowly):

```
class StickFigureSprite(Sprite):
    def __init__(self, game):
        Sprite.__init__(self, game)
     ❶ self.images_left = [
            PhotoImage(file='figure-L1.gif'),
            PhotoImage(file='figure-L2.gif'),
            PhotoImage(file='figure-L3.gif')
        ]
     ❷ self.images_right = [
            PhotoImage(file='figure-R1.gif'),
            PhotoImage(file='figure-R2.gif'),
            PhotoImage(file='figure-R3.gif')
        ]
     ❸ self.image = game.canvas.create_image(200, 470,
                image=self.images_left[0], anchor='nw')
```

This code loads the three left images, which we'll use to animate the stick figure running left, and the three right images, which we'll use to animate the stick figure running right.

We create the object variables `images_left` ❶ and `images _right` ❷ . Each contains a list of the `PhotoImage` objects we created in Chapter 10 , showing the stick figure facing left and right.

We draw the first image ❸ with `images_left[0]` using the canvas's `create_image` function at position (`200, 470`), which puts the stick figure in the middle of the game screen, at the bottom of the canvas. The `create_image` function returns a number that identifies the image on the canvas. We store this identifier in the object variable `image` for later use.

SETTING UP VARIABLES

The next part of the `__init__` function sets up more variables we'll use later in this code:

```
        self.image = game.canvas.create_image(200, 470,
                image=self.images_left[0], anchor='nw')
     ❶ self.x = -2
     ❷ self.y = 0
        self.current_image = 0
        self.current_image_add = 1
        self.jump_count = 0
        self.last_time = time.time()
        self.coordinates = Coords()
```

The object variables `x` ❶ and `y` ❷ will store the amount we'll be adding to the stick figure's horizontal (*x1* and *x2*) or vertical (*y1* and *y2*) coordinates when he is moving around the screen.

As you learned in Chapter 11 , to animate something with the `tkinter` module, we add values to the object's `x` or `y` position to move it around the canvas. By setting `x` to –2 and `y` to 0, we subtract 2 from the `x` position later in the code and add nothing to the vertical position, to make the stick figure run to the left.

Note

Remember that a negative `x` number means move left on the canvas, and a positive `x` number means move right. A negative `y` number means move up, and a positive `y` number means move down.

Next, we create the object variable `current_image` to store the image's index position as currently displayed on the screen. Our list of left-facing images, `images_left` , contains *figure-L1.gif* , *figure-L2.gif* , and *figure-L3.gif* . Those are index positions 0, 1, and 2.

The `current_image_add` variable will contain the number we'll add to that index position stored in `current_image` to get the next index position. For example, if the image at index position 0 is displayed, we add 1 to get the next image at index position 1, and then add 1 again to get the final image in the list at index position 2. (You'll see how we use this variable for animation in the next chapter.)

The `jump_count` variable is a counter we'll use while the stick figure is jumping. The `last_time` variable will record the last time we changed the image when animating our stick figure. We store the current time using the `time` function of the `time` module.

Finally, we set the `coordinates` object variable to an object of the `Coords` class, with no initialization parameters set (`x1` , `y1` , `x2` , and `y2` are all 0).

Unlike with the platforms, the stick figure's coordinates will change, so we'll set these values later.

BINDING TO KEYS

In the final part of the `__init__` function, the `bind` functions bind a key to something in our code that needs to be run when the key is pressed:

```
        self.jump_count = 0
        self.last_time = time.time()
        self.coordinates = Coords()
        game.canvas.bind_all('<KeyPress-Left>', self.turn_left)
        game.canvas.bind_all('<KeyPress-Right>', self.turn_right)
        game.canvas.bind_all('<space>', self.jump)
```

We bind `<KeyPress-Left>` to the `turn_left` function, `<KeyPress-Right>` to the `turn_right` function, and `<space>` to the `jump` function. Now we need to create those functions to make the stick figure move.

TURNING THE STICK FIGURE LEFT AND RIGHT

The `turn_left` and `turn_right` functions ensure the stick figure is not jumping, and then set the value of the object variable `x` to move him left and right. (Our game doesn't allow us to change his direction in midair.)

```
        game.canvas.bind_all('<KeyPress-Left>', self.turn_left)
        game.canvas.bind_all('<KeyPress-Right>', self.turn_right)
        game.canvas.bind_all('<space>', self.jump)

    def turn_left(self, evt):
        if self.y == 0:
         ❶ self.x = -2

    def turn_right(self, evt):
        if self.y == 0:
         ❷ self.x = 2
```

Python calls the `turn_left` function when the player presses the left arrow key, and it passes an object with information about what the player did as a parameter. This object is called an *event object* , and we give it the parameter name `evt` .

Note

The event object isn't important for our purposes, but we need to include it as a parameter of our functions or we'll get an error, because Python is expecting it to be there. The event object contains things like the x *and* y *positions of the mouse (for a mouse event), a code identifying a particular key that has been pressed (for keyboard events), and other information. For this game, none of that information is useful, so we can safely ignore it.*

To see if the stick figure is jumping, we check the value of the `y` object variable. If the value is not 0, the stick figure is jumping. In this code, if the value of `y` is 0, we set `x` to –2 to run left ❶ or we set it to 2 to run right ❷ . We use –2 and 2, because setting the value to –1 or 1 wouldn't make the stick figure move across the screen fast enough.

Once you have the animation working for your stick figure, try changing this value to see what difference it makes.

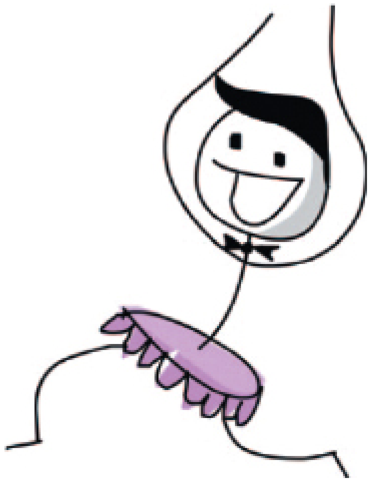
MAKING THE STICK FIGURE JUMP

The jump function is very similar to the turn_left and turn_right functions:

```
def turn_right(self, evt):
    if self.y == 0:
        self.x = 2

def jump(self, evt):
    if self.y == 0:
        self.y = -4
        self.jump_count = 0
```

This function again takes an evt parameter (the event object), which we can ignore because we don't need any more information about the event (same as before). If this function is called, we know the spacebar was pressed.

Because we want our stick figure to jump only if he is not already jumping, we check to see if y is equal to 0. If the stick figure is not jumping, we set y to –4 (to move him vertically up the screen), and set jump_count to 0. We'll use jump_count to make sure the stick figure doesn't keep jumping forever. Instead, we'll let him jump for a specific count and then have him come back down again, as if gravity were pulling him. We'll add this code in the next chapter.



WHAT WE HAVE SO FAR

Let's review the definitions of the classes and functions in our game thus far, and where they should be in your file.

At the top of your program, you should have your `import` statements, followed by the `Game` and `Coords` classes. The `Game` class will be used to create an object that will be the main controller for our game, and objects of the `Coords` class are used to hold the positions of things in our game (like the platforms and Mr. Stick Man):

```
from tkinter import *
import random
import time

class Game:
    --snip--
class Coords:
    --snip--
```

Next, you should have the `within` functions (which tell whether the coordinates of one sprite are within the same area of another sprite), the `Sprite` parent class (which is the parent class of all the sprites in our game), the `PlatformSprite` class, and the beginning of the `StickFigureSprite` class. We used the `PlatformSprite` class to create platform objects, which our stick figure will jump across. We also created one object of the `StickFigureSprite` class, to represent the main character in our game:

```
def within_x(co1, co2):
    --snip--
def within_y(co1, co2):
    --snip--
def collided_left(co1, co2):
    --snip--
def collided_right(co1, co2):
    --snip--
def collided_top(co1, co2):
    --snip--
def collided_bottom(y, co1, co2):
    --snip--
class Sprite:
    --snip--
class PlatformSprite(Sprite):
    --snip--
class StickFigureSprite(Sprite):
    --snip--
```

Finally, at the end of your program, you should have code that creates all the objects in our game so far: the game object itself and the platforms. The final line is where we call the `mainloop` function:

```
g = Game()
platform1 = PlatformSprite(g, PhotoImage(file='platform1.gif'),
                           0, 480, 100, 10)
...
g.sprites.append(platform1)
...
g.mainloop()
```

If your code looks a bit different, or you're having trouble getting it working, you can always skip ahead to the end of Chapter 16 for the full listing of the entire game.

WHAT YOU LEARNED

In this chapter, we began working on the class for our stick figure. At the moment, if we created an object of this class, it wouldn't really do much besides load the images it needs for animating the stick figure and set up a few object variables to be used later in the code. This class contains a couple of functions for changing the values in those object variables based on keyboard events (when the player presses the left or right arrow or the spacebar).

In the next chapter, we'll finish our game. We'll write the functions for the `StickFigureSprite` class to display and animate the stick figure and move him around the screen. We'll also add the exit (the door) that Mr. Stick Man is trying to reach.

16
COMPLETING THE MR. STICK MAN GAME

In the previous three chapters, we've been developing our game: *Mr. Stick Man Races for the Exit* . We created the graphics, and then wrote code to add the background image, platforms, and stick figure. In this chapter, we'll fill in the missing pieces to animate the stick figure and add the door. You'll find the full listing for the complete game at the end of this chapter. If you get lost or become confused when writing some of this code, compare your code with that listing to see where you might have gone wrong.

ANIMATING THE STICK FIGURE

So far, we've created a basic class for our stick figure, loading the images we'll be using and binding keys to some functions. But none of our code will do anything particularly interesting if you run the game at this point.

Now we'll add the remaining functions to the `StickFigureSprite` class we created in Chapter 15 : `animate` , `move` , and `coords` . The `animate` function will draw the different stick figure images; `move` will determine where the character needs to move to; and `coords` will return the stick figure's current position. (Unlike with the platform sprites, we need to recalculate the position of the stick figure as he moves around the screen.)

CREATING THE ANIMATE FUNCTION

First, we'll add the `animate` function, which will need to check for movement and change the image accordingly.

CHECKING FOR MOVEMENT

We don't want to change the stick figure image too quickly in our animation, or its movement won't look realistic. Think about a flip animation, drawn in the corner of a notepad—if you flip the pages too quickly, you may not get the full effect of what you've drawn.

The first half of the `animate` function checks to see if the stick figure is running to the left or right, and then uses the `last_time` variable to decide whether to change the current image. This variable will help us control the speed of our animation. The function will go after the `jump` function, which we added to our `StickFigureSprite` class in Chapter 15 (page 238):

```
    def animate(self):
        if self.x != 0 and self.y == 0:
            if time.time() - self.last_time > 0.1:
                self.last_time = time.time()
                self.current_image += self.current_image_add
                if self.current_image >= 2:
                    self.current_image_add = -1
                if self.current_image <= 0:
                    self.current_image_add = 1
```

In the first `if` statement, we check to see if `x` is not 0 to determine whether the stick figure is moving (either left or right), and we check to see if `y` is 0 to determine that the stick figure is not jumping. If this `if` statement is `True` , we need to animate our stick figure; if not, he's standing still, so there's no need to keep drawing. If the stick figure isn't moving, we drop out of the function, and the rest of the code in this listing is ignored.

We then calculate the amount of time since the `animate` function was last called, by subtracting the value of the `last_time` variable from the current time, using `time.time()` . This calculation is used to decide whether to draw the next image in the sequence. If the result is greater than a tenth of a second (0.1), we continue with the block of code. We set the `last_time` variable to the current time, basically resetting the stopwatch to start timing again for the next image change.

Next, we add the value of the object variable `current_image_add` to the variable `current_image` , which stores the index position of the currently displayed image. Remember that we created the `current_image_add` variable in the stick figure's `__init__` function in Chapter 15 (see page 235), so when the `animate` function is first called, the value of the variable has already been set to 1.

Then, we check to see if the value of the index position in `current_image` is greater than or equal to 2; if so, we change the value of `current_image_add` to –1. The process is similar for the last two lines; once we reach 0, we need to start counting up again.

Note

If you're having trouble figuring out how to indent this code, here's a hint: there are 8 spaces at the line beginning `if self.x` *and 20 spaces at the last line.*

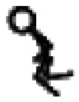
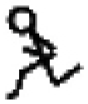
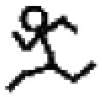
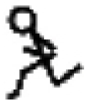
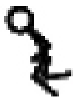
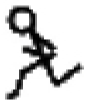
To help you understand what's going on in the function so far, imagine that you have a sequence of colored blocks in a line on the floor. You move your finger from one block to the next, and each block that your finger points to has a number (1, 2, 3, 4, and so on)—this is the `current_image` variable. The number of the block your finger moves to (it points at one block at a time) is the number stored in the `current_image_add` variable. When your finger moves one way up the line of blocks, you're adding 1 each time, and when it hits the end of the line and moves back down, you're subtracting 1 (adding –1).

The code we've added to our `animate` function performs this process, but instead of colored blocks, we have the three stick figure images for each direction stored in a list. The index positions of these images are 0, 1, and 2. As we animate the stick figure, once we reach the last image, we start counting down, and once we reach the first image, we need to start counting up again. As a result, we create the effect of a running figure.

Table 16-1 shows how we move through the list of images, using the index positions we calculate in the `animate` function.

Table 16-1: Image Positions in Animation

Position 0	**Position 1**	**Position 2**	**Position 1**	**Position 0**	**Position 1**
Counting up	Counting up	Counting up	Counting down	Counting down	Counting up

Position 0	Position 1	Position 2	Position 1	Position 0	Position 1
					

CHANGING THE IMAGE

In the next half of the `animate` function, we change the currently displayed image, using the calculated index position:

```
    def animate(self):
    ...
        if self.x < 0:
            if self.y != 0:
                self.game.canvas.itemconfig(self.image,
                        image=self.images_left[2])
         ❶ else:
                self.game.canvas.itemconfig(self.image,
                        image=self.images_left[self.current_image])
        elif self.x > 0:
            if self.y != 0:
                self.game.canvas.itemconfig(self.image,
                        image=self.images_right[2])
            else:
                self.game.canvas.itemconfig(self.image,
                        image=self.images_right[self.current_image])
```

Firstly, if `x` is less than 0, the stick figure is moving left, so Python moves into the block of code, which checks whether `y` is not equal to 0 (meaning the stick figure is jumping). If `y` is not equal to 0 (the stick figure is jumping), we use the canvas's `itemconfig` function to change the displayed image to the last image in our list of left-facing images at `images_left[2]` . Because the stick figure is jumping, we'll use the image showing him in full stride to make the animation look a bit more realistic, as you can see in Figure 16-1 .

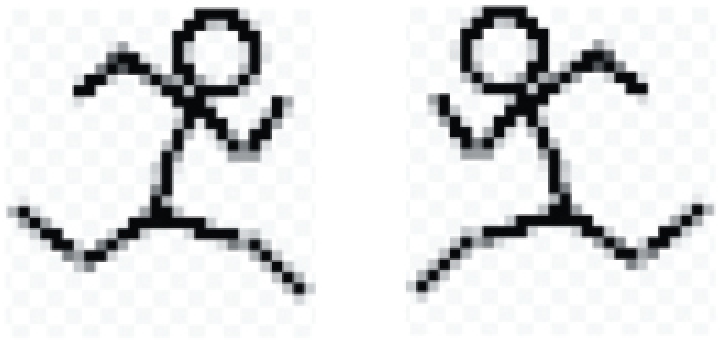


Figure 16-1: Jumping images

If the stick figure is not jumping (that is, `y` is equal to 0), the `else` block ❶ uses `itemconfig` to change the displayed image to whatever index position is in the variable `current_image` .

At the `elif` statement, we see if the stick figure is running right (`x` is greater than 0), and Python moves into the code block. This code is very similar to the first block, again checking whether the stick figure is jumping and drawing the correct image if so, except that it uses the `images_right` list.

GETTING THE STICK FIGURE'S POSITION

Because we'll need to determine where the stick figure is on the screen (since he's moving around), the `coords` function will differ from the other `Sprite` class functions. We'll use the `coords` function of the canvas to determine where the stick figure is, and then use those values to set the `x1` , `y1` and `x2` , `y2` values of the `coordinates` variable we created in the `__init__` function at the beginning of Chapter 15 . Add the following code after the `animate` function:

```
    def coords(self):
        xy = self.game.canvas.coords(self.image)
        self.coordinates.x1 = xy[0]
        self.coordinates.y1 = xy[1]
        self.coordinates.x2 = xy[0] + 27
        self.coordinates.y2 = xy[1] + 30
        return self.coordinates
```

When we created the `Game` class in Chapter 14 , one of the object variables was the `canvas` . We use the `coords` function of this `canvas` variable (with `self.game.canvas.coords`), which takes the identifier of something drawn on the canvas, and returns the *x* and *y* positions as a list of two numbers. In this case, we use the identifier stored in the variable `current_image` and store the returned list in the variable `xy` . We then use the two values to set the coordinates for our stick figure. The value `xy[0]` (that's the first number in the list) becomes our `x1` coordinate, and the value `xy[1]` (the second number in the list) becomes our `y1` coordinate. So that's the top-left position of the figure.

Because all of the stick figure images we created are 27 pixels wide by 30 pixels high, we can determine what the `x2` and `y2` variables should be (that's the bottom-right position of the figure) by adding the width and the height to the `xy[0]` and `xy[1]` values, respectively.

So, if `self.game.canvas.coords(self.image)` returns `[270, 350]` we will end up with the following values:

- `self.coordinates.x1` will be 270
- `self.coordinates.y1` will be 350
- `self.coordinates.x2` will be 297
- `self.coordinates.y2` will be 380

Finally, on the last line of the function, we return the object variable `coordinates` that we just updated.

MAKING THE STICK FIGURE MOVE

The final function of the `StickFigureSprite` class, `move` , is in charge of actually moving our game character around the screen. It also needs to be able to tell us when the character has bumped into something.

STARTING THE MOVE FUNCTION

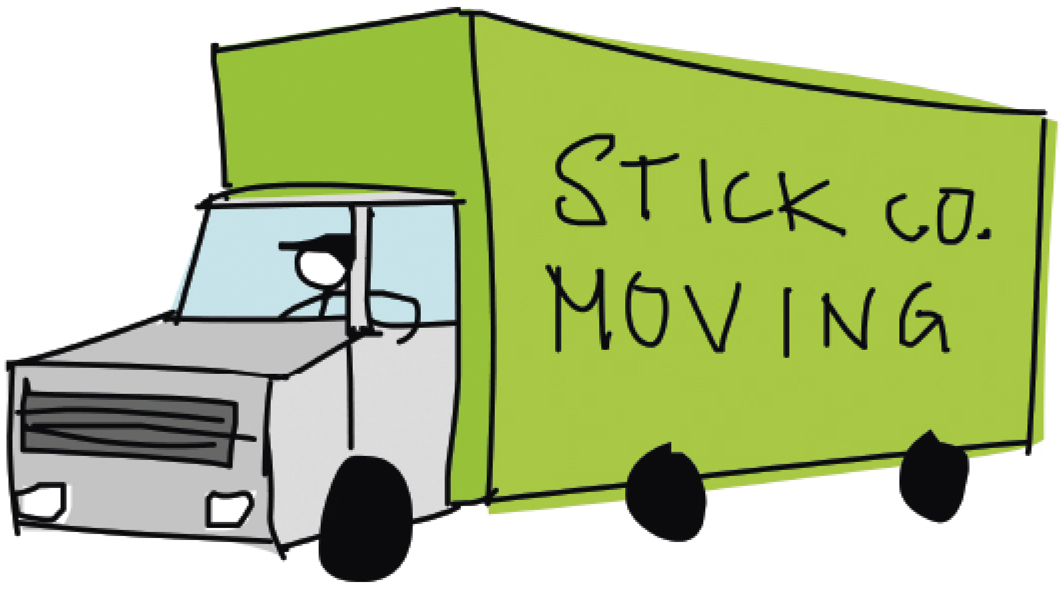
The following code is for the first part of the `move` function. This will go after `coords` :

```
    def move(self):
        self.animate()
        if self.y < 0:
            self.jump_count += 1
            if self.jump_count > 20:
                self.y = 4
        if self.y > 0:
            self.jump_count -= 1
```

The first line (`self.animate()`) calls the function we created earlier in this chapter, which changes the currently displayed image if necessary. Then, we see whether the value of `y` is less than 0. If it is, we know that the stick figure is jumping because a negative value will move him up the screen. (Remember that 0 is at the top of the canvas, and the bottom of the canvas is pixel position 500.)

Next, we add 1 to `jump_count` . We want our stick figure to jump up, but not to keep floating up the screen forever (it's jumping, after all), so we use that variable to count the number of times we have executed the `move` function—if it reaches 20, we should change `y` to 4 to start the stick figure falling again.

We then see if the value of `y` is greater than 0 (meaning the character must be falling); if it is, we subtract 1 from `jump_count` because once we've counted up to 20, we need to count back down again. (Move your hand slowly up in the air while counting to 20, and then move it back down again while counting down from 20, and you'll get a sense of how calculating the stick figure jumping up and down is supposed to work.)



In the next few lines of the `move` function, we call the `coords` function, which tells us where our character is on the screen, and then store its result in the `co` variable. We then create the variables `left` , `right` , `top` , `bottom` , and `falling` . We'll use each in the remainder of this function:

```
        co = self.coords()
        left = True
        right = True
        top = True
        bottom = True
        falling = True
```

Notice that each variable has been set to the Boolean value `True` . We'll use these as indicators to check whether the character has hit something on the screen or is falling.

HAS THE STICK FIGURE HIT THE BOTTOM OR TOP OF THE CANVAS?

The next section of the `move` function checks whether our character has hit the bottom or top of the canvas. Add the following code:

```
        if self.y > 0 and co.y2 >= self.game.canvas_height:
            self.y = 0
            bottom = False
        elif self.y < 0 and co.y1 <= 0:
```

```
        self.y = 0
        top = False
```

If the character is falling down the screen, `y` will be greater than 0, so we need to make sure it hasn't yet hit the bottom of the canvas (or it will vanish off the bottom of the screen). To do so, we see if its *y2* position (the bottom of the stick figure) is greater than or equal to the `canvas_height` variable of the game object. If it is, we set the value of `y` to 0 to stop the stick figure from falling, and then set the `bottom` variable to `False` , which tells the remaining code that we no longer need to see if the stick figure has hit the bottom.

The process of determining whether the stick figure has hit the top of the screen is very similar to the way we determine whether he has hit the bottom. To do so, we first see if the stick figure is jumping (`y` is less than 0), and then we see if his *y1* position is less than or equal to 0, meaning he has hit the top of the canvas. If both conditions are true, we set `y` equal to 0 to stop the movement. We also set the `top` variable to `False` to tell the remaining code that we no longer need to see if the stick figure has hit the top.

HAS THE STICK FIGURE HIT THE SIDE OF THE CANVAS?

We follow almost exactly the same process as in the preceding code to determine whether the stick figure has hit the right and left sides of the canvas, as follows:

```
    if self.x > 0 and co.x2 >= self.game.canvas_width:
        self.x = 0
        right = False
    elif self.x < 0 and co.x1 <= 0:
        self.x = 0
        left = False
```

The `if` statement is based on the fact that we know the stick figure is running to the right if `x` is greater than 0. We also know whether he has hit the right-hand side of the screen by seeing if the *x2* position (`co.x2`) is greater than or equal to the width of the canvas stored in `canvas_width` .

If either statement is true, we set `x` equal to 0 (to stop the stick figure from running) and set the `right` or `left` variables to `False` .

COLLIDING WITH OTHER SPRITES

Once we've determined whether the figure has hit the sides of the screen, we need to see if he has hit anything else on the screen. We use the following code to loop through the list of sprite objects stored in the `game` object to see if the stick figure has hit any of them:

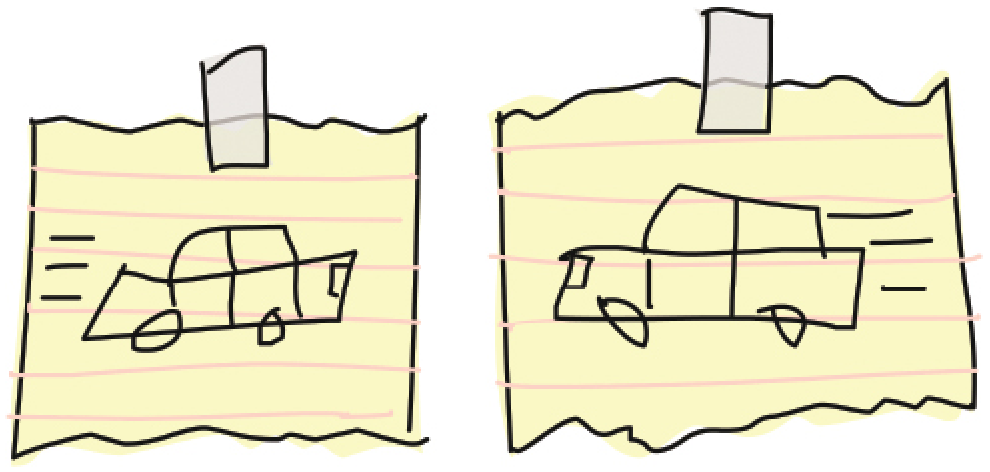
```
        for sprite in self.game.sprites:
            if sprite == self:
                continue
            sprite_co = sprite.coords()
            if top and self.y < 0 and collided_top(co, sprite_co):
                self.y = -self.y
                top = False
```

At the `for` statement, we loop through the list of sprites, assigning each one in turn to the variable `sprite` . Then we say that if the sprite is equal to `self` (that's another way of saying, "if this particular sprite is actually me"), we don't need to determine whether the stick figure has collided because he would have only hit himself. If the `sprite` variable is equal to `self` , we use `continue` to jump to the next sprite in the list (`continue` simply tells Python to ignore the rest of the code in the block and continue the loop).

Next, we get the coordinates of the new sprite by calling its `coords` function and storing the results in the `sprite_co` variable.

The final `if` statement checks for the following:

- The stick figure has not hit the top of the canvas (the `top` variable is still true).
- The stick figure is jumping (the `y` value is less than 0).
- The top of the stick figure has collided with the sprite from the list (using the `collided_top` function we created on page 224).



If all of these conditions are true, we want the sprite to start falling back down again, so we reverse the value of the `y` variable (`self.y` becomes `-self.y`). The `top` variable is set to `False` because once the stick figure has hit the top, we don't need to keep checking for a collision.

COLLIDING AT THE BOTTOM

The next part of the loop checks to see if the bottom of our character has hit something:

```
        if bottom and self.y > 0 and collided_bottom(self.y,
                co, sprite_co):
            self.y = sprite_co.y1 - co.y2
            if self.y < 0:
                self.y = 0
            bottom = False
            top = False
```

We start with three similar checks: whether the `bottom` variable is still set, whether the character is falling (`y` is greater than 0), and whether the bottom of our character has hit the sprite. If all three checks are true, we subtract the bottom *y* value (`y2`) of the stick figure from the top *y* value of the sprite (`y1`). This might seem strange, so let's discuss why we do this.

Imagine that our game character has fallen off a platform. He moves down the screen 4 pixels each time the `mainloop` function runs, until the foot of the stick figure is 3 pixels above another platform. Let's say the stick figure's bottom (`y2`) is at position 57, and the top of the platform (

`y1`) is at position 60. In this case, the `collided_bottom` function would return `True` , because its code will add the value of `y` (which is 4) to the stick figure's `y2` variable, resulting in 61.

However, we don't want Mr. Stick Man to stop falling as soon as it looks like he'll hit a platform or the bottom of the screen, because that would be like taking a huge jump off a step and stopping in midair, an inch above the ground. That may be a neat trick, but it won't look right in our game. Instead, if we subtract the character's `y2` value (57) from the platform's `y1` value (60), we get 3, the amount the stick figure should drop in order to land properly on top of the platform.

We continue by making sure the calculation doesn't result in a negative number (`if self.y < 0:`); if it does, we set `y` equal to 0. (If we let the number be negative, the stick figure would fly back up again, and we don't want that to happen.)

Finally, we set the `top` and `bottom` flags to `False` , so we no longer need to check whether the stick figure has collided at the top or bottom with another sprite.

We'll follow this code with one more "bottom" check to see whether the stick figure has run off the edge of a platform. Here's the code for this `if` statement:

```
        if bottom and falling and self.y == 0 \
                and co.y2 < self.game.canvas_height \
                and collided_bottom(1, co, sprite_co):
            falling = False
```

For the `falling` variable to be set to `False` , we must check that the following five elements are all true:

- The `bottom` flag is set to `True` .
- The stick figure should be falling (the `falling` flag is still set to `True`).
- The stick figure isn't already falling (`y` is 0).
- The bottom of the sprite hasn't hit the bottom of the screen (it's less than the canvas height).

- The stick figure has hit the top of a platform (`collided_bottom` returns `True`).

Then we set the `falling` variable to `False` to stop the figure from dropping down the screen.

Note

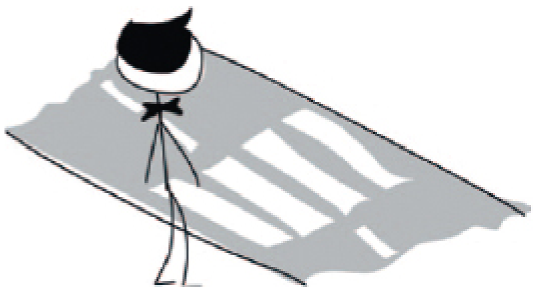
You can check whether the value of a Boolean variable is True in an `if` statement by simply referencing the variable. For example, `if bottom == True and falling == True` can be rewritten simply as `if bottom and falling` (as we did above).

CHECKING LEFT AND RIGHT

We've checked whether the stick figure has hit a sprite at the bottom or the top. Now we need to check whether he has hit the left or right side, with this code:

```
            if left and self.x < 0 and collided_left(co, sprite_co):
                self.x = 0
                left = False
            if right and self.x > 0 and collided_right(co, sprite_co):
                self.x = 0
                right = False
```

First, we see if we should still be looking for collisions to the left (`left` is still set to `True`) and whether the stick figure is moving to the left (`x` is less than 0). We also check to see if the stick figure has collided with a sprite by using the `collided_left` function. If these three conditions are true, we set `x` equal to 0 (to make the stick figure stop running), and set `left` to `False` , so that we no longer check for collisions on the left.



The code is similar for collisions to the right. We set `x` equal to 0 again, and set `right` to `False` , to stop checking for right-hand collisions.

Now, with checks for collisions in all four directions, our `for` loop should look like this:

```
        for sprite in self.game.sprites:
            if sprite == self:
                continue
            sprite_co = sprite.coords()
            if top and self.y < 0 and collided_top(co, sprite_co):
                self.y = -self.y
                top = False
            if bottom and self.y > 0 and collided_bottom(self.y,
                    co, sprite_co):
                self.y = sprite_co.y1 - co.y2
                if self.y < 0:
                    self.y = 0
                bottom = False
                top = False
            if bottom and falling and self.y == 0 \
                    and co.y2 < self.game.canvas_height \
                    and collided_bottom(1, co, sprite_co):
                falling = False
            if left and self.x < 0 and collided_left(co, sprite_co):
                self.x = 0
                left = False
            if right and self.x > 0 and collided_right(co, sprite_co):
                self.x = 0
                right = False
```

We need to add only a few more lines to the `move` function, as follows:

```
        if falling and bottom and self.y == 0 \
                and co.y2 < self.game.canvas_height:
            self.y = 4
        self.game.canvas.move(self.image, self.x, self.y)
```

We check whether both the `falling` and `bottom` variables are set to `True` . If so, we've looped through every platform sprite in the list without colliding at the bottom.

The final check in this line determines whether the bottom of our character is less than the canvas height—that is, above the ground (the bottom of the canvas). If the stick figure hasn't collided with anything and is above the ground, he is standing in midair, so he should start falling (in other words, he has run off the end of a platform). To make him run off the end of any platform, we set `y` equal to 4.

Lastly, we move the image across the screen, according to the values we set in the variables `x` and `y` . The fact that we've looped through the sprites checking for collisions may mean that we've set both variables to 0, because the stick figure has collided on the left and with the bottom. In that case, the call to the `move` function of the canvas will actually do nothing.

It may also be the case that Mr. Stick Man has walked off the edge of a platform. If that happens, `y` will be set to 4 and Mr. Stick Man will fall downward.

Phew, that was a long function!

TESTING OUR STICK FIGURE SPRITE

Having created the `StickFigureSprite` class, let's try it out by adding the following two lines just before the call to the `mainloop` function:

```
sf = StickFigureSprite(g)
g.sprites.append(sf)
```

We create a `StickFigureSprite` object and label it with the `sf` variable. As we did with the platforms, we add this new variable to the list of sprites stored in the game object.

Now run the program. You should find that Mr. Stick Man can run, jump from platform to platform, and fall!

THE EXIT!

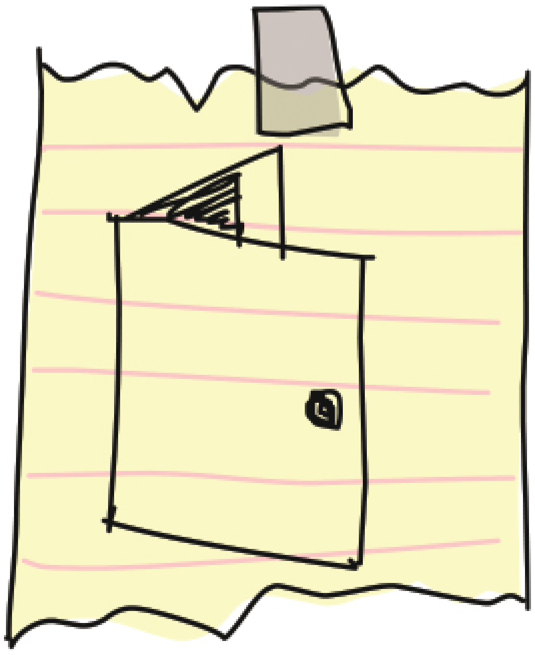
The only thing missing from our game is the exit. We'll finish up by creating a sprite for the door, adding code to detect the door, and giving our program a door object.

CREATING THE DOORSPRITE CLASS

We need to create one more class: `DoorSprite` . The start of the code is as follows:

```
class DoorSprite(Sprite):
    def __init__(self, game, photo_image, x, y, width, height):
        Sprite.__init__(self, game)
        self.photo_image = photo_image
        self.image = game.canvas.create_image(x, y,
                image=self.photo_image, anchor='nw')
        self.coordinates = Coords(x, y, x + (width / 2), y + height)
        self.endgame = True
```

The `__init__` function of the `DoorSprite` class has parameters for `self` , a `game` object, a `photo_image` object, the `x` and `y` coordinates, and the `width` and `height` of the image. We call `Sprite.__init__` as with our other sprite classes.



We then save the parameter `photo_image` using an object variable with the same name, as we did with `PlatformSprite` . We create a display image using the canvas `create_image` function and save the identifying number returned by that function using the object variable `image` .

Next, we set the coordinates of `DoorSprite` to the `x` and `y` parameters (which become the *x1* and *y1* positions of the door), and then calculate the *x2* and *y2* positions. We calculate the *x2* position by adding half of the width (the `width` variable divided by 2) to the `x` parameter. For example, if `x` is 10 (the `x1` coordinate is also 10) and the `width` is 40, the `x2` coordinate would be 30 (10 plus half of 40).

Why use this confusing little calculation? Because, unlike with the platforms, where we want Mr. Stick Man to stop running as soon as he collides with the side of the platform, we want him to stop in front of the door. You'll see this in action when you play the game and make it to the door.

Unlike the *x1* position, the *y1* position is simple to calculate. We just add the value of the `height` variable to the `y` parameter, and that's it.

Finally, we set the `endgame` object variable to `True` . This says that when the stick figure reaches the door, the game ends.

DETECTING THE DOOR

Now we need to change the code in the `StickFigureSprite` class of the `move` function that determines when the stick figure has collided with a sprite on the left or the right. Here's the first change:

```
            if left and self.x < 0 and collided_left(co, sprite_co):
                self.x = 0
                left = False
                if sprite.endgame:
                    self.game.running = False
```

We check to see if the stick figure has collided with a sprite that has an `endgame` variable set to `True` . If it does, we set the `running` variable to `False` , and everything stops—we've reached the end of the game.

We'll add these same lines to the code that checks for a collision on the right. Here's the code:

```
            if right and self.x > 0 and collided_right(co, sprite_co):
                self.x = 0
                right = False
                if sprite.endgame:
                    self.game.running = False
```

ADDING THE DOOR OBJECT

Our final addition to the game code is an object for the door. We'll add this before the main loop. Just before creating the stick figure object, we'll create a door object and then add it to the list of sprites. Here's the code:

```
g.sprites.append(platform7)
g.sprites.append(platform8)
g.sprites.append(platform9)
g.sprites.append(platform10)
door = DoorSprite(g, PhotoImage(file='door1.gif'), 45, 30, 40, 35)
g.sprites.append(door)
sf = StickFigureSprite(g)
g.sprites.append(sf)
g.mainloop()
```

We create a door object using the variable for our game object, `g` , followed by a `PhotoImage` (the door image we created in Chapter 13). We set the `x` and `y` parameters to 45 and 30 to put the door on a platform near the top of the screen, and set the width and height to 40 and 35. We add the door object to the list of sprites, as with all the other sprites in the game.

You can see the result when Mr. Stick Man reaches the door. He stops running in front of the door, rather than next to it, as shown in Figure 16-2 .

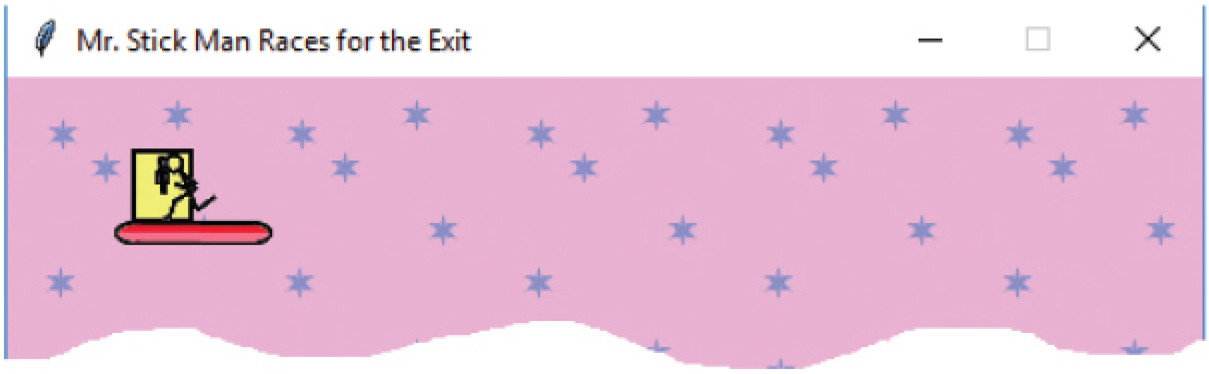


Figure 16-2: Reaching the door

THE FINAL GAME

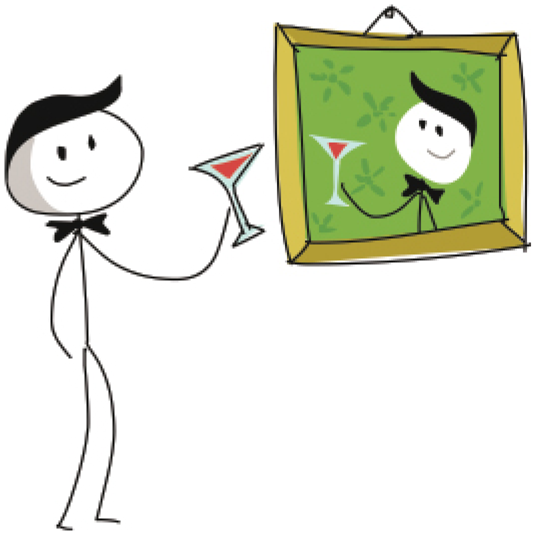
The full listing of our game is now a bit more than 200 lines of code. The following is the complete code for the game. If you have trouble getting your game to work, compare each function (and each class) to this listing:

```
from tkinter import *
import random
import time

class Coords:
    def __init__(self, x1=0, y1=0, x2=0, y2=0):
        self.x1 = x1
        self.y1 = y1
        self.x2 = x2
        self.y2 = y2

def within_x(co1, co2):
    if (co1.x1 > co2.x1 and co1.x1 < co2.x2) \
            or (co1.x2 > co2.x1 and co1.x2 < co2.x2) \
            or (co2.x1 > co1.x1 and co2.x1 < co1.x2) \
            or (co2.x2 > co1.x1 and co2.x2 < co1.x2):
        return True
    else:
        return False

def within_y(co1, co2):
    if (co1.y1 > co2.y1 and co1.y1 < co2.y2) \
            or (co1.y2 > co2.y1 and co1.y2 < co2.y2) \
            or (co2.y1 > co1.y1 and co2.y1 < co1.y2) \
            or (co2.y2 > co1.y1 and co2.y2 < co1.y2):
        return True
    else:
        return False

def collided_left(co1, co2):
    if within_y(co1, co2):
        if co1.x1 >= co2.x1 and co1.x1 <= co2.x2:
            return True
    return False

def collided_right(co1, co2):
    if within_y(co1, co2):
        if co1.x2 >= co2.x1 and co1.x2 <= co2.x2:
            return True
    return False

def collided_top(co1, co2):
    if within_x(co1, co2):
        if co1.y1 >= co2.y1 and co1.y1 <= co2.y2:
            return True
    return False

def collided_bottom(y, co1, co2):
    if within_x(co1, co2):
        y_calc = co1.y2 + y
        if y_calc >= co2.y1 and y_calc <= co2.y2:
            return True
    return False

class Sprite:
    def __init__(self, game):
        self.game = game
        self.endgame = False
        self.coordinates = None
    def move(self):
        pass
    def coords(self):
        return self.coordinates

class PlatformSprite(Sprite):
    def __init__(self, game, photo_image, x, y, width, height):
        Sprite.__init__(self, game)
        self.photo_image = photo_image
        self.image = game.canvas.create_image(x, y,
                image=self.photo_image, anchor='nw')
        self.coordinates = Coords(x, y, x + width, y + height)

class StickFigureSprite(Sprite):
    def __init__(self, game):
        Sprite.__init__(self, game)
        self.images_left = [
            PhotoImage(file='figure-L1.gif'),
            PhotoImage(file='figure-L2.gif'),
            PhotoImage(file='figure-L3.gif')
        ]
        self.images_right = [
            PhotoImage(file='figure-R1.gif'),
            PhotoImage(file='figure-R2.gif'),
            PhotoImage(file='figure-R3.gif')
        ]
        self.image = game.canvas.create_image(200, 470,
                image=self.images_left[0], anchor='nw')
        self.x = -2
        self.y = 0
        self.current_image = 0
        self.current_image_add = 1
        self.jump_count = 0
        self.last_time = time.time()
        self.coordinates = Coords()
        game.canvas.bind_all('<KeyPress-Left>', self.turn_left)
        game.canvas.bind_all('<KeyPress-Right>', self.turn_right)
        game.canvas.bind_all('<space>', self.jump)

    def turn_left(self, evt):
        if self.y == 0:
            self.x = -2

    def turn_right(self, evt):
        if self.y == 0:
            self.x = 2

    def jump(self, evt):
        if self.y == 0:
            self.y = -4
            self.jump_count = 0

    def animate(self):
        if self.x != 0 and self.y == 0:
            if time.time() - self.last_time > 0.1:
                self.last_time = time.time()
                self.current_image += self.current_image_add
                if self.current_image >= 2:
                    self.current_image_add = -1
                if self.current_image <= 0:
                    self.current_image_add = 1
        if self.x < 0:
            if self.y != 0:
                self.game.canvas.itemconfig(self.image,
                        image=self.images_left[2])
            else:
                self.game.canvas.itemconfig(self.image,
                        image=self.images_left[self.current_image])
        elif self.x > 0:
            if self.y != 0:
                self.game.canvas.itemconfig(self.image,
                        image=self.images_right[2])
            else:
                self.game.canvas.itemconfig(self.image,
                        image=self.images_right[self.current_image])

    def coords(self):
        xy = self.game.canvas.coords(self.image)
        self.coordinates.x1 = xy[0]
        self.coordinates.y1 = xy[1]
        self.coordinates.x2 = xy[0] + 27
        self.coordinates.y2 = xy[1] + 30
        return self.coordinates

    def move(self):
        self.animate()
        if self.y < 0:
            self.jump_count += 1
            if self.jump_count > 20:
                self.y = 4
        if self.y > 0:
            self.jump_count -= 1
        co = self.coords()
        left = True
        right = True
        top = True
        bottom = True
        falling = True
        if self.y > 0 and co.y2 >= self.game.canvas_height:
            self.y = 0
            bottom = False
        elif self.y < 0 and co.y1 <= 0:
            self.y = 0
            top = False
        if self.x > 0 and co.x2 >= self.game.canvas_width:
            self.x = 0
            right = False
        elif self.x < 0 and co.x1 <= 0:
            self.x = 0
            left = False

        for sprite in self.game.sprites:
            if sprite == self:
                continue
            sprite_co = sprite.coords()
            if top and self.y < 0 and collided_top(co, sprite_co):
                self.y = -self.y
                top = False
            if bottom and self.y > 0 and collided_bottom(self.y,
                    co, sprite_co):
                self.y = sprite_co.y1 - co.y2
                if self.y < 0:
                    self.y = 0
                bottom = False
                top = False
            if bottom and falling and self.y == 0 \
                    and co.y2 < self.game.canvas_height \
                    and collided_bottom(1, co, sprite_co):
                falling = False
            if left and self.x < 0 and collided_left(co, sprite_co):
                self.x = 0
                left = False
                if sprite.endgame:
                    self.game.running = False
            if right and self.x > 0 and collided_right(co, sprite_co):
                self.x = 0
                right = False
                if sprite.endgame:
                    self.game.running = False

        if falling and bottom and self.y == 0 \
                and co.y2 < self.game.canvas_height:
            self.y = 4
        self.game.canvas.move(self.image, self.x, self.y)

class DoorSprite(Sprite):
    def __init__(self, game, photo_image, x, y, width, height):
        Sprite.__init__(self, game)
        self.photo_image = photo_image
        self.image = game.canvas.create_image(x, y,
                image=self.photo_image, anchor='nw')
        self.coordinates = Coords(x, y, x + (width / 2), y + height)
        self.endgame = True

class Game:
    def __init__(self):
        self.tk = Tk()
        self.tk.title('Mr. Stick Man Races for the Exit')
        self.tk.resizable(0, 0)
        self.tk.wm_attributes('-topmost', 1)
        self.canvas = Canvas(self.tk, width=500, height=500,
                             highlightthickness=0)
        self.canvas.pack()
        self.tk.update()
        self.canvas_height = self.canvas.winfo_height()
        self.canvas_width = self.canvas.winfo_width()
        self.bg = PhotoImage(file='background.gif')
        w = self.bg.width()
        h = self.bg.height()
        for x in range(0, 5):
            for y in range(0, 5):
                self.canvas.create_image(x * w, y * h,
                        image=self.bg, anchor='nw')
        self.sprites = []
        self.running = True

    def mainloop(self):
        while True:
            if self.running == True:
                for sprite in self.sprites:
                    sprite.move()
            self.tk.update_idletasks()
            self.tk.update()
            time.sleep(0.01)

g = Game()
platform1 = PlatformSprite(g, PhotoImage(file='platform1.gif'),
                           0, 480, 100, 10)
platform2 = PlatformSprite(g, PhotoImage(file='platform1.gif'),
                           150, 440, 100, 10)
platform3 = PlatformSprite(g, PhotoImage(file='platform1.gif'),
                           300, 400, 100, 10)
platform4 = PlatformSprite(g, PhotoImage(file='platform1.gif'),
                           300, 160, 100, 10)
platform5 = PlatformSprite(g, PhotoImage(file='platform2.gif'),
                           175, 350, 66, 10)
platform6 = PlatformSprite(g, PhotoImage(file='platform2.gif'),
                           50, 300, 66, 10)
platform7 = PlatformSprite(g, PhotoImage(file='platform2.gif'),
                           170, 120, 66, 10)
platform8 = PlatformSprite(g, PhotoImage(file='platform2.gif'),
                           45, 60, 66, 10)
platform9 = PlatformSprite(g, PhotoImage(file='platform3.gif'),
                           170, 250, 32, 10)
platform10 = PlatformSprite(g, PhotoImage(file='platform3.gif'),
                            230, 200, 32, 10)
g.sprites.append(platform1)
g.sprites.append(platform2)
g.sprites.append(platform3)
g.sprites.append(platform4)
g.sprites.append(platform5)
g.sprites.append(platform6)
g.sprites.append(platform7)
g.sprites.append(platform8)
g.sprites.append(platform9)
g.sprites.append(platform10)
door = DoorSprite(g, PhotoImage(file='door1.gif'), 45, 30, 40, 35)
```

```
g.sprites.append(door)
sf = StickFigureSprite(g)
g.sprites.append(sf)
g.mainloop()
```

WHAT YOU LEARNED

In this chapter, we completed our game, *Mr. Stick Man Races for the Exit* . We created a class for our animated stick figure and wrote functions to move him around the screen and animate him as he moves (changing from one image to the next to give the illusion of running). We've used basic collision detection to tell when he has hit the left or right sides of the canvas, and when he has hit another sprite, such as a platform or a door. We've also added collision code to tell when he hits the top of the screen or the bottom, and to make sure that when he runs off the edge of a platform, he tumbles down accordingly. We added code to tell when Mr. Stick Man has reached the door, so the game comes to an end.



PROGRAMMING PUZZLES

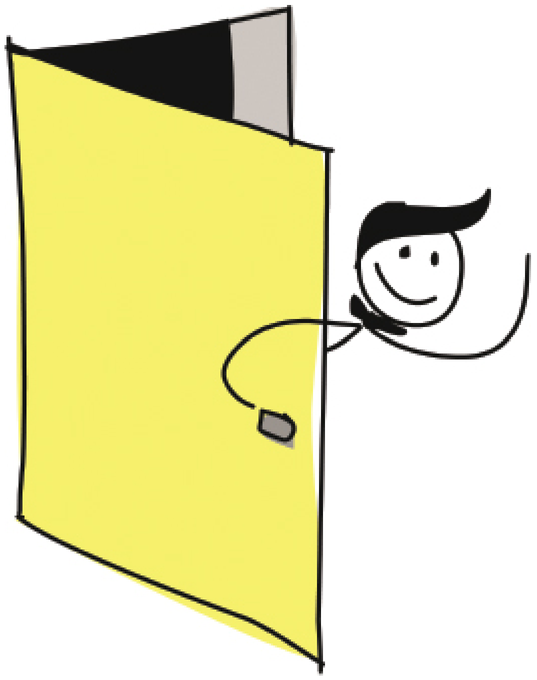
There's a lot more we can do to improve the game. We can add code to make it more professional looking and more interesting to play. Try adding the following features and then compare your code with the solutions at *http://python-for-kids.com* .

#1: "YOU WIN!"

Like the "Game Over" text in the *Bounce!* game we completed in Chapter 12 , add "You Win!" text when the stick figure reaches the door.

#2: ANIMATING THE DOOR

In Chapter 13 , we created two images for the door: one open and one closed. When Mr. Stick Man reaches the door, the door image should change to the open door, Mr. Stick Man should vanish, and the door image should revert to the closed door. This will give the illusion that Mr. Stick Man is exiting and closing the door as he leaves. You can do this by changing the `DoorSprite` class and the `StickFigureSprite` class.



#3: MOVING PLATFORMS

Try adding a new class called `MovingPlatformSprite` . This platform should move from side to side, making it more difficult for Mr. Stick Man to reach the door at the top. You can pick some platforms to be moving, and leave some platforms to be static, depending on how hard you want your game to be.

#4: LAMP AS A SPRITE

Instead of the bookshelf and lamp we added as background images in Chapter 14 's third programming puzzle, try adding a lamp that the stick man has to jump over. Rather than being a part of the game's background, it will be a sprite similar to the platforms or the door.

AFTERWORD: WHERE TO GO FROM HERE

You've learned some fundamental programming concepts in your tour of Python, but there is plenty more to discover—either with Python or using other programming languages. While Python is incredibly useful, it is not always the best tool for every task, so do not be afraid to try other ways to program your computer.

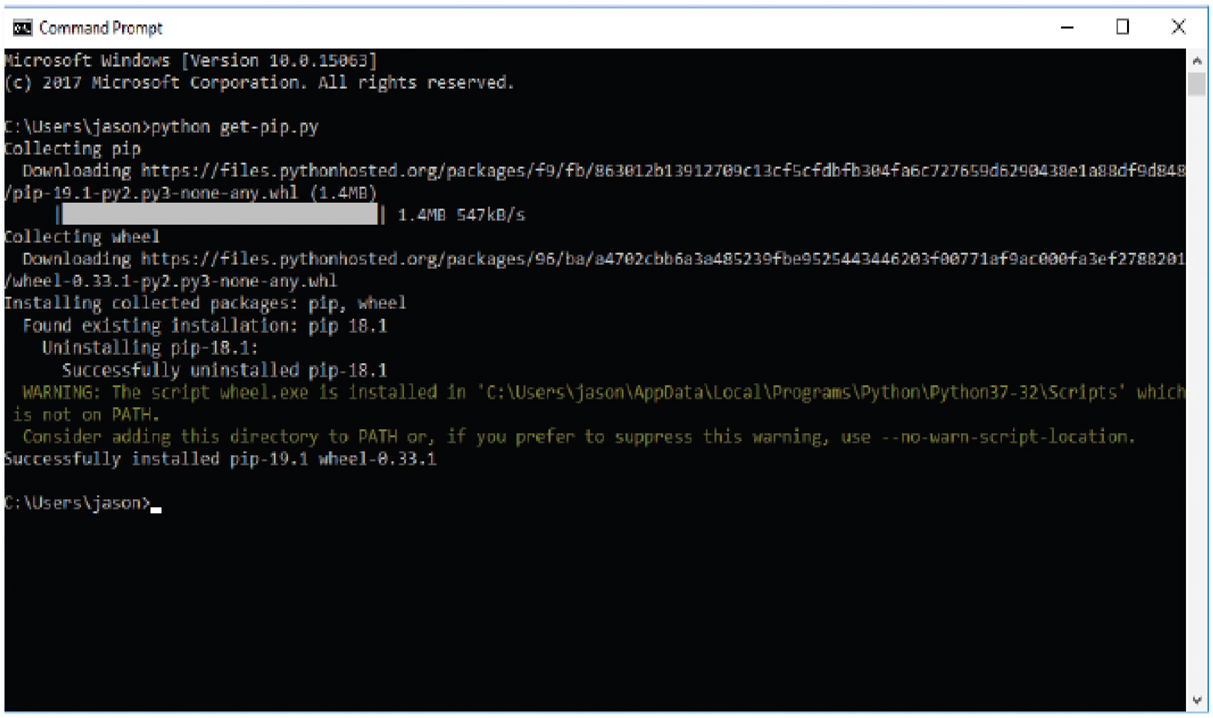
If you want to stick with Python, and are looking for more advanced books to read, the Python wiki (books page) is a good place to start: *https://wiki.python.org/moin/PythonBooks/* .

If you just want to explore more of what Python can do, it has a lot of built-in modules. (This is Python's "batteries included" philosophy. Check the Python documentation for the full list of what's available: *https://docs.python.org/3/py-modindex.html* .) In addition, there are a huge number of modules that are provided for free by programmers from all over the world. For example, you could try *Pygame* (*https://www.pygame.org*) for games development, or *Jupyter Notebooks* (*https://jupyter.org*), a web-based environment for editing and running Python code in the browser. These and other modules are available to browse at *https://pypi.org* . To install these modules, you'll need to use a tool called `pip` , which we'll briefly look at next.

INSTALLING PYTHON PIP ON WINDOWS

As long as you've installed Python 3.10 or newer, `pip` should be installed by default. To install `pip` on Windows, if you're using an older version of Python, you can download the script *get-pip.py* from *https://bootstrap.pypa.io/get-pip.py* .

Save the file to your home folder and then open a command prompt (click **Start** and enter **`cmd`** in the search box). To install, type **`python get-pip.py`** .



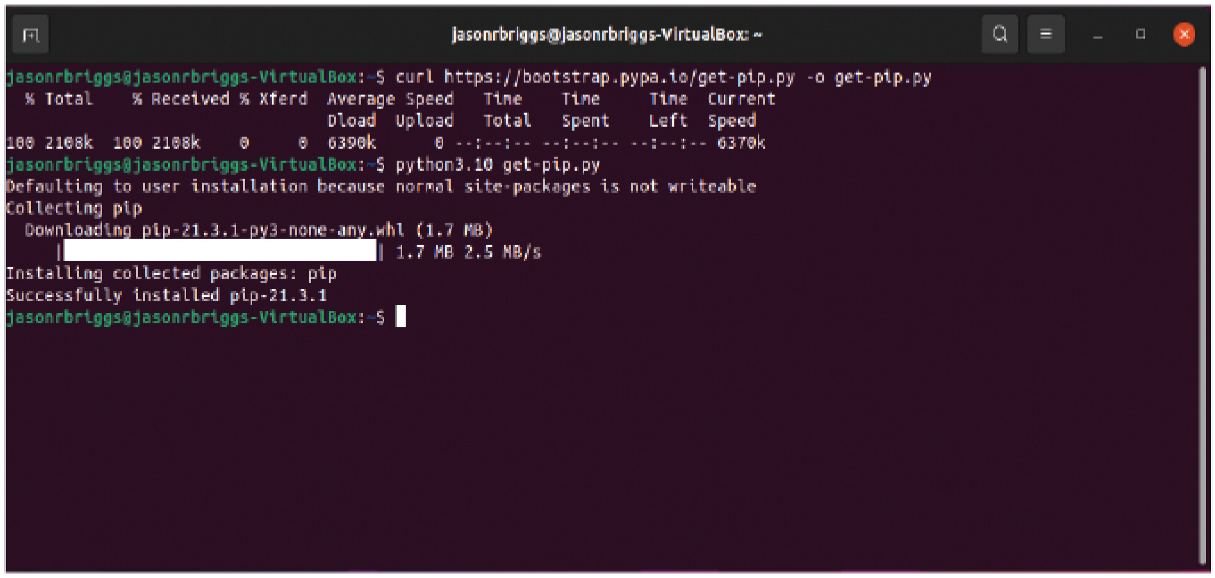
```
Microsoft Windows [Version 10.0.15063]
(c) 2017 Microsoft Corporation. All rights reserved.

C:\Users\jason>python get-pip.py
Collecting pip
  Downloading https://files.pythonhosted.org/packages/f9/fb/863012b13912709c13cf5cfdbfb304fa6c727659d6290438e1a88df9d848
/pip-19.1-py2.py3-none-any.whl (1.4MB)
     |████████████████████████████████| 1.4MB 547kB/s
Collecting wheel
  Downloading https://files.pythonhosted.org/packages/96/ba/a4702cbb6a3a485239fbe9525443446203f00771af9ac000fa3ef2788201
/wheel-0.33.1-py2.py3-none-any.whl
Installing collected packages: pip, wheel
  Found existing installation: pip 18.1
    Uninstalling pip-18.1:
      Successfully uninstalled pip-18.1
  WARNING: The script wheel.exe is installed in 'C:\Users\jason\AppData\Local\Programs\Python\Python37-32\Scripts' which
 is not on PATH.
  Consider adding this directory to PATH or, if you prefer to suppress this warning, use --no-warn-script-location.
Successfully installed pip-19.1 wheel-0.33.1

C:\Users\jason>_
```

INSTALLING PYTHON PIP ON UBUNTU

To install `pip` on Ubuntu Linux, you'll need the system administrator password. Open a terminal and enter the following commands (be sure to change the version number in the following command as needed):

```
sudo apt install python3.10-distutils
sudo apt install curl
curl https://bootstrap.pypa.io/get-pip.py -o get-pip.py
python3.10 get-pip.py
```

With the first two commands, you may get an error saying they are already installed; this can be safely ignored.



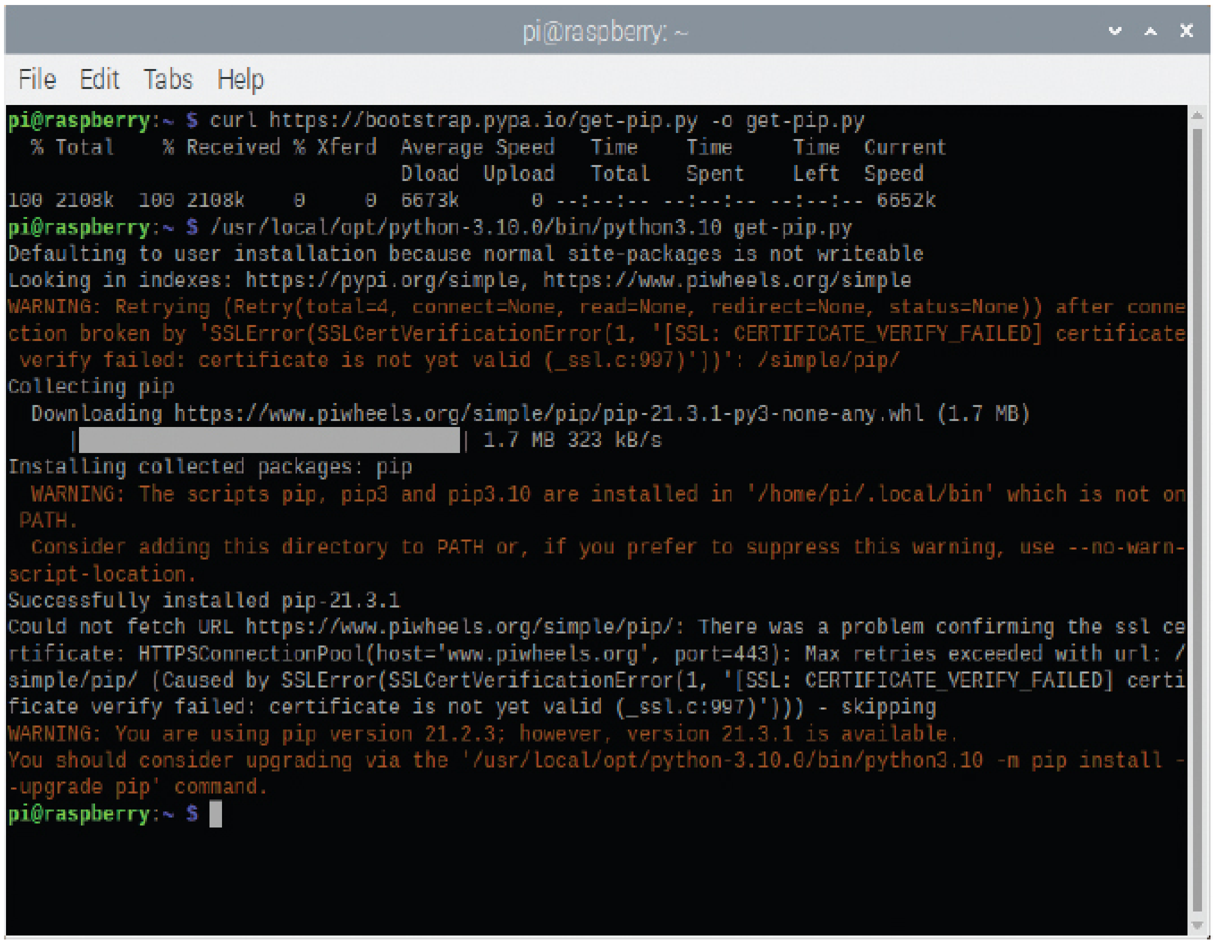
```
jasonrbriggs@jasonrbriggs-VirtualBox:~$ curl https://bootstrap.pypa.io/get-pip.py -o get-pip.py
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100 2108k  100 2108k    0     0  6390k      0 --:--:-- --:--:-- --:--:-- 6370k
jasonrbriggs@jasonrbriggs-VirtualBox:~$ python3.10 get-pip.py
Defaulting to user installation because normal site-packages is not writeable
Collecting pip
  Downloading pip-21.3.1-py3-none-any.whl (1.7 MB)
     |████████████████████████████████| 1.7 MB 2.5 MB/s
Installing collected packages: pip
Successfully installed pip-21.3.1
jasonrbriggs@jasonrbriggs-VirtualBox:~$ 
```

INSTALLING PYTHON PIP ON RASPBERRY PI

To install `pip` on Raspberry Pi, open a terminal and enter the following commands:

```
curl https://bootstrap.pypa.io/get-pip.py -o get-pip.py
python3.10 get-pip.py
```

You can safely ignore any warning messages that display.



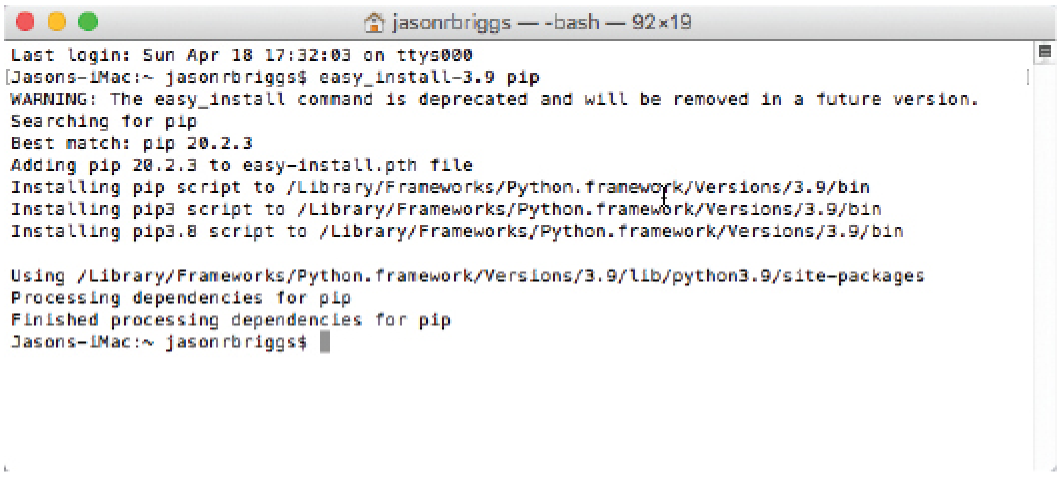
```
pi@raspberry:~ $ curl https://bootstrap.pypa.io/get-pip.py -o get-pip.py
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100 2108k  100 2108k    0     0  6673k      0 --:--:-- --:--:-- --:--:-- 6652k
pi@raspberry:~ $ /usr/local/opt/python-3.10.0/bin/python3.10 get-pip.py
Defaulting to user installation because normal site-packages is not writeable
Looking in indexes: https://pypi.org/simple, https://www.piwheels.org/simple
WARNING: Retrying (Retry(total=4, connect=None, read=None, redirect=None, status=None)) after conne
ction broken by 'SSLError(SSLCertVerificationError(1, '[SSL: CERTIFICATE_VERIFY_FAILED] certificate
 verify failed: certificate is not yet valid (_ssl.c:997)'))': /simple/pip/
Collecting pip
  Downloading https://www.piwheels.org/simple/pip/pip-21.3.1-py3-none-any.whl (1.7 MB)
     |████████████████████████████████| 1.7 MB 323 kB/s
Installing collected packages: pip
  WARNING: The scripts pip, pip3 and pip3.10 are installed in '/home/pi/.local/bin' which is not on
 PATH.
  Consider adding this directory to PATH or, if you prefer to suppress this warning, use --no-warn-
script-location.
Successfully installed pip-21.3.1
Could not fetch URL https://www.piwheels.org/simple/pip/: There was a problem confirming the ssl ce
rtificate: HTTPSConnectionPool(host='www.piwheels.org', port=443): Max retries exceeded with url: /
simple/pip/ (Caused by SSLError(SSLCertVerificationError(1, '[SSL: CERTIFICATE_VERIFY_FAILED] certi
ficate verify failed: certificate is not yet valid (_ssl.c:997)'))) - skipping
WARNING: You are using pip version 21.2.3; however, version 21.3.1 is available.
You should consider upgrading via the '/usr/local/opt/python-3.10.0/bin/python3.10 -m pip install -
-upgrade pip' command.
pi@raspberry:~ $ 
```

INSTALLING PYTHON PIP ON MACOS

As long as you've installed Python 3.10 or newer, `pip` should be installed by default. If you're using an older version of Python, you might need to install `pip` by opening a terminal and running the following command:

```
easy_install-3.9 pip
```

Depending on which version of Python you have installed, you might need to enter a different version of `easy_install` (such as `easy_install-3.7` , for example).



```
Last login: Sun Apr 18 17:32:03 on ttys000
Jasons-iMac:~ jasonrbriggs$ easy_install-3.9 pip
WARNING: The easy_install command is deprecated and will be removed in a future version.
Searching for pip
Best match: pip 20.2.3
Adding pip 20.2.3 to easy-install.pth file
Installing pip script to /Library/Frameworks/Python.framework/Versions/3.9/bin
Installing pip3 script to /Library/Frameworks/Python.framework/Versions/3.9/bin
Installing pip3.8 script to /Library/Frameworks/Python.framework/Versions/3.9/bin

Using /Library/Frameworks/Python.framework/Versions/3.9/lib/python3.9/site-packages
Processing dependencies for pip
Finished processing dependencies for pip
Jasons-iMac:~ jasonrbriggs$
```

TRYING OUT PYGAME

To get an idea of how *PyGame* works, first install it using `pip` . Open a command prompt (on Windows, enter `cmd` in the search box; on Ubuntu, Raspberry Pi, or macOS, search for `Terminal`) and then enter the following command (your version number may be different):

```
pip3.10 install pygame
```

Note

Depending on your Windows and Python versions, this might not work. If you get an error such as this:

```
'pip' is not recognized as an internal or external command,
operable program or batch file.
```

then try running the following commands:

```
cd %HOMEPATH%
AppData\Local\Programs\Python\Python310\python -m pip install pygame
```

Writing code with `PyGame` is a bit more complicated than using `tkinter` . For example, in Chapter 10 , we displayed an image by using `tkinter` with this code:

```
from tkinter import *
tk = Tk()
canvas = Canvas(tk, width=400, height=400)
canvas.pack()
myimage = PhotoImage(file='c:\\Users\\jason\\test.gif')
canvas.create_image(0, 0, anchor=NW, image=myimage)
```

To display an image with `PyGame` , you can use the following:

```
import pygame
pygame.init()
display = pygame.display.set_mode((500, 500))
img = pygame.image.load('c:\\Users\\jason\\test.gif')
display.blit(img, (0, 0))
pygame.display.flip()
```

After importing the `pygame` module, we call the `init` function; this initializes the module so it can be used. We then set up the display, passing a tuple for the width and height (500 pixels wide by 500 pixels high). Note the additional parentheses here are important: first, parentheses for the function itself (`set_mode(...)`), and then parentheses for a tuple (`500, 500`).

We then load the image, the reference (or label) for which is the variable `img` (remember, depending on which operating system you're using, you might need to change the path of that image). Next, we draw the image to the display using the `blit` function, passing the `img` variable and a tuple containing the top-left position of the image (`0, 0`). The image won't show in the window yet; the next function on the last line —where we `flip` the display—actually causes the image to appear. This line is effectively telling `Pygame` to redraw the display window.

Note

If you run this outside of IDLE, you'll need to add a few additional lines at the end:

```
while True:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            raise SystemExit
```

This code is to stop the window from immediately closing after displaying the image.

OTHER GAMES AND GRAPHICS PROGRAMMING

If you want to do more with games or graphics programming, you'll find that many options are available, not just with Python. Here are just a few:

- Scratch (*http://scratch.mit.edu*), a tool for developing games and animations. (It's a block-based visual programming language, so quite different from programming with Python.)
- Construct3 (*https://www.construct.net*), a commercial tool for creating games in a browser.
- Game Maker Studio (*https://www.yoyogames.com*), another commercial tool for creating games.
- Godot (*https://godotengine.org*), a free games engine for creating both 2D and 3D graphics.
- Unity (*http://unity.com*), another commercial tool for creating games.
- Unreal Engine (*https://www.unrealengine.com*), another commercial tool for creating games.

An online search will uncover a wealth of resources to help you get started with any of these options, or at least show you what is possible if you carry on with programming in the future.

OTHER PROGRAMMING LANGUAGES

If you're interested in other programming languages, consider checking out some of the most popular: JavaScript, Java, C#, C, C++, Ruby, Go, Rust, and Swift (although there are a lot more). We'll take a brief tour of these languages and see how a Hello World program (like the Python version we started with in Chapter 1) would look in each one. Note that none of these languages are specifically intended for beginning programmers, and most are significantly different from Python.

JAVASCRIPT

JavaScript (*https://dev.java/*) is a programming language typically used for web pages, game programming, and other activities. You can easily create a simple *HTML* page (the language used to create web pages) that contains a JavaScript program and run it inside a browser without needing a shell, command line, or anything else.

A "Hello World" example in JavaScript will be different depending on whether you run it in a browser or shell. In a shell, you can type the following:

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

In a browser, it might look like this:

```
<html>
    <body>
        <script type="text/javascript">
            alert("Hello World");
        </script>
    </body>
</html>
```

JAVA

Java (*http://www.oracle.com/technetwork/java/index.html*) is a moderately complicated programming language with a large built-in library of modules (called *packages*). Java is the programming language used on Android mobile devices, and you can use it on most operating systems.

Here's an example of printing "Hello World" in Java:

```
public class HelloWorld {
    public static final void main(String[] args) {
        System.out.println("Hello World");
    }
}
```

C#

C# (*https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide*), pronounced "C sharp," is a moderately complicated programming language for Windows that is very similar to Java and JavaScript in syntax. It's part of Microsoft's .NET platform.

Here's an example of printing "Hello World" in C#:

```
public class Hello
{
   public static void Main()
   {
      System.Console.WriteLine("Hello World");
   }
}
```

C/C++

C (*http://www.cprogramming.com*) and C++ (*http://www.stroustrup.com/C++.html*) are complicated programming languages used on all operating systems. You'll find both free and commercial versions available. Both languages (though perhaps C++ more than C) have a steep learning curve (in other words, they're not necessarily good for beginners). For example, you'll find that you need to manually code some features that Python provides (like telling the computer that you need to use a chunk of memory to store an object). Many commercial games and game consoles are programmed in some form of C or C++.

An example of printing "Hello World" in C is as follows:

```
#include <stdio.h>
int main ()
{
  printf ("Hello World\n");
}
```

An example in C++ might look like this:

```
#include <iostream>
int main()
{
  std::cout << "Hello World\n";
  return 0;
}
```

RUBY

Ruby (*http://www.ruby-lang.org*) is a free programming language available on all major operating systems. It's mostly used for creating websites, specifically using the framework *Ruby on Rails* . (A *framework* is a set of libraries supporting the development of specific types of applications.)

Here's an example of printing "Hello World" in Ruby:

```
puts "Hello World"
```

GO

Go (*https://golang.org*) is a programming language similar to C, but slightly simpler.

You can print "Hello World" with Go like so:

```
package main
import "fmt"
func main() {
        fmt.Println("Hello World")
}
```

RUST

Rust (*https://www.rust-lang.org*) is a language originally developed by Mozilla Research (the same people who make the Firefox browser).

A simple program for printing "Hello World" with Rust might look like this:

```
fn main() {
    println!("Hello World")
}
```

SWIFT

Swift (*https://swift.org*) is a language developed by Apple for its devices (iOS, macOS, and so on), so it's most suitable if you're using an Apple product.

We could print "Hello World" with Swift like this:

```
import Swift
print("Hello World")
```

FINAL WORDS

Whether you stick with Python or decide to try out another programming language, you should still find the concepts that you've discovered in this book useful. Even if you don't continue with computer programming, understanding some of the fundamental ideas can help with all sorts of activities, whether in school or later on, at work.

Good luck and have fun with your programming!

A
B
C

A
PYTHON KEYWORDS

Keywords in Python (and most programming languages) are words that have special meaning. They are used as part of the programming language itself, and therefore must not be used for anything else. For example, if you try to use keywords as variables, or use them in the wrong way, you'll get strange error messages from the Python console. This appendix describes each of the Python keywords. You should find this to be a handy reference as you continue to program.

AND

The keyword `and` is used to join two expressions together in a statement (like an `if` statement) to say that both expressions must be true. Here's an example:

```
if age > 12 and age < 20:
    print('Beware the teenager!!!!')
```

This code means that the value of the variable `age` must be greater than `12` and less than `20` before the message will be printed.

AS

The keyword `as` can be used to give another name to an imported module. For example, suppose you had a module with a very long name:

`i_am_a_python_module_that_is_not_very_useful` .

It would be enormously annoying to need to type this module name every time you wanted to use it:

```
>>> import i_am_a_python_module_that_is_not_very_useful
>>> i_am_a_python_module_that_is_not_very_useful.do_something()

I have done something that is not useful.
>>> i_am_a_python_module_that_is_not_very_useful.do_something_else()

I have done something else that is not useful!!
```

Instead, you can give the module a new, shorter name when you import it, and then simply use that new name (like a nickname), as follows:

```
>>> import i_am_a_python_module_that_is_not_very_useful as notuseful
>>> notuseful.do_something()

I have done something that is not useful.
>>> notuseful.do_something_else()

I have done something else that is not useful!!
```

ASSERT

The `assert` keyword is used to say that a value must be true. It’s another way of catching errors and problems in code, usually in more advanced programs (which is why we don’t use `assert` in *Python for Kids*). Here’s a simple `assert` statement:

```
>>> mynumber = 10
>>> assert mynumber < 5
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
```

```
    assert mynumber < 5
AssertionError
```

In this example, we assert that the value of the variable `mynumber` is less than 5. It isn't, so Python displays an error (called an `AssertionError`).

ASYNC

The `async` keyword is used to define something called a *native coroutine* . This is an advanced concept used in asynchronous programming (which is doing multiple things in parallel, or doing things after some time).

AWAIT

The `await` keyword is also used for asynchronous programming (similar to `async`).

BREAK

The `break` keyword is used to stop some code from running. You might use `break` inside a `for` loop, like this:

```
age = 10
for x in range(1, 100):
    print(f'counting {x}')
    if x == age:
        print('end counting')
        break
```

Since the variable `age` is set to 10 here, this code will print out the following:

```
counting 1
counting 2
counting 3
counting 4
counting 5
counting 6
counting 7
counting 8
counting 9
counting 10
end counting
```

Once the value of the variable `x` reaches 10, the code prints the text "end counting" and then breaks out of the loop.

CLASS

The keyword `class` is used to define a type of object, like a vehicle, animal, or person. Classes can have a function called `__init__` , which is used to perform all the tasks an object of the class needs when it is created. For example, an object of the `Car` class might need a `color` variable when it's created:

```
>>> class Car:
        def __init__(self, color):
            self.color = color

>>> car1 = Car('red')
>>> car2 = Car('blue')
>>> print(car1.color)
red
>>> print(car2.color)
blue
```

CONTINUE

The `continue` keyword is a way to "jump" to the next item in a loop so the remaining code in the loop block is not executed. Unlike `break` , we don't jump out of the loop—we just carry on with the next item. For example, if we had a list of items and wanted to skip items starting with *b* , we could use the following code:

```
>>> my_items = ['apple', 'aardvark', 'banana', 'badger',
        'clementine', 'camel']
>>> for item in my_items:
        if item.startswith('b'):
            continue
        print(item)

apple
aardvark
clementine
camel
```

We first create our list of items, and then use a `for` loop to loop over the items and run a block of code for each. If the item starts with the letter *b* , we continue to the next item. Otherwise, we print out the item.

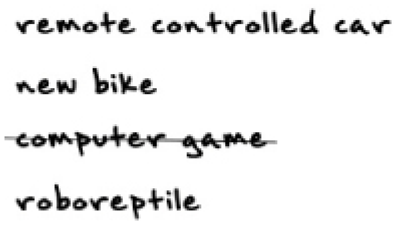
DEF

The `def` keyword is used to define a function. For example, we can create a function to convert a number of years into the equivalent number of minutes:

```
>>> def minutes(years):
        return years * 365 * 24 * 60
>>> minutes(10)
5256000
```

DEL

The `del` keyword is used to delete something. For example, if you had a list of things you wanted for your birthday in your diary, but then changed your mind about one of them, you might cross it off the list and add something new:



remote controlled car

new bike

~~computer game~~

roboreptile

In Python, the original list would look like this:

```
what_i_want = ['remote controlled car', 'new bike', 'computer game']
```

You could remove the computer game by using `del` and the index of the item you want to delete. You could then add the new item with the `append` function:

```
del what_i_want[2]
what_i_want.append('roboreptile')
```

And then print the new list:

```
print(what_i_want)
['remote controlled car', 'new bike', 'roboreptile']
```

ELIF

The keyword `elif` is used as part of an `if` statement. See the description of the `if` keyword for an example.

ELSE

The keyword `else` is used as part of an `if` statement. See the description of the `if` keyword for an example.

EXCEPT

The `except` keyword is used for catching problems in fairly complicated code.

FINALLY

The keyword `finally` is used to make sure that if an error occurs, certain code runs (usually to tidy up any mess that a piece of code has left behind). This keyword is for more advanced programming.

FOR

The for keyword is used to create a loop of code that runs a certain number of times. Here's an example:

```
for x in range(0, 5):
    print(f'x is {x}')
```

This for loop executes the block of code (the print statement) five times, resulting in the following output:

```
x is 0
x is 1
x is 2
x is 3
x is 4
```

FROM

When importing a module, you can import just the part you need using the from keyword. For example, the turtle module introduced in Chapter 4 has a class called Turtle , which we use to create a Turtle object (which includes the canvas on which the turtle moves). Here's how we import the entire turtle module and then use the Turtle class:

```
import turtle
t = turtle.Turtle()
```

You could also import the Turtle class on its own, and then use it directly (without referring to the turtle module at all):

```
from turtle import Turtle
t = Turtle()
```

You might do this so the next time you look at the top of that program, you can see all the functions and classes you're using (which is particularly useful in larger programs that import a lot of modules). However, if you choose to do this, you won't be able to use the parts of the module you haven't imported. For example, the time module has functions called `localtime` and `gmtime` , but if you import only `localtime` and then try to use `gmtime` , you'll get an error:

```
>>> from time import localtime
>>> print(localtime())
(2019, 1, 30, 20, 53, 42, 1, 30, 0)
>>> print(gmtime())
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'gmtime' is not defined
```

The error message `name 'gmtime' is not defined` means that Python doesn't know anything about the function `gmtime` , which is because you haven't imported it.

If a particular module has multiple functions that you want to use, and you don't want to refer to them by using module names (for example, `time.localtime` , or `time.gmtime`), you can import everything in the module by using an asterisk (`*`), like this:

```
>>> from time import *
>>> print(localtime())
(2021, 1, 30, 20, 57, 7, 1, 30, 0)
>>> print(gmtime())
(2021, 1, 30, 13, 57, 9, 1, 30, 0)
```

This form imports everything from the `time` module, and you can now refer to the individual functions by name.

GLOBAL

The idea of *scope* in programs is introduced in Chapter 7 . Scope refers to the visibility of a variable. If a variable is defined outside a function, it can usually be seen (it's visible) inside the function. On the other hand, if the variable is defined inside a function, usually it can't be seen outside that function. The `global` keyword is one exception to this rule. A variable defined as global can be seen everywhere. Here's an example:

```
>>> def test():
        global a
        a = 1
        b = 2
```

What do you think happens when you call `print(a)` and then `print(b)` after running the function test? The first will work, but the second will display an error message:

```
>>> test()
>>> print(a)
1
>>> print(b)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'b' is not defined
```

The variable `a` has been changed to have global scope inside the function, so it's visible, even once the function has completed; but `b` is still visible only inside the function. (You must use the `global` keyword before setting the value of your variable.)

IF

The `if` keyword is used to make a decision about something. It can also be used with the keywords `else` and `elif` (else if). An `if` statement is a way of saying, "If something is true, then perform an action of some kind." Here's an example:

```
if toy_price > 1000:
    print('That toy is overpriced')
elif toy_price > 100:
    print('That toy is expensive')
else:
    print('I can afford that toy')
```

This `if` statement says that if a toy price is over $1,000, display a message that it is overpriced; otherwise, if the toy price is over $100, display a message that it's expensive. If neither of those conditions is true, it should display the message, "I can afford that toy."

IMPORT

The `import` keyword tells Python to load a module so it can be used. For example, the following code tells Python to use the `sys` module:

```
import sys
```

IN

The `in` keyword is used in expressions to see if an item is within a collection of items. For example, can the number 1 be found in a list (a collection) of numbers?

```
>>> if 1 in [1,2,3,4]:
        print('number is in the list')
number is in the list
```

Here's how to find out if the string `'pants'` is in a list of clothing items:

```
>>> clothing_list = ['shorts', 'undies', 'boxers', 'long johns',
                 'knickers']
>>> if 'pants' in clothing_list:
        print('pants is in the list')
    else:
        print('where are my pants?')
where are my pants?
```

IS

The `is` keyword is a bit like the *equal to* operator (`==`), which is used to tell if two things are equal (for example, `10 == 10` is true, and `10 == 11` is false). However, there is a fundamental difference between `is` and `==` . If you are comparing two objects (such as lists), `==` may return `true` , while `is` may not (even if you think the objects are the same). This is an advanced programming concept.

LAMBDA

The `lambda` keyword is used to create anonymous, or inline, functions. This keyword is used in more advanced programs.

NONLOCAL

The `nonlocal` keyword is used to include a variable in the scope of a function when it's declared outside the function. This keyword is used in more advanced programs.

NOT

If something is true, the `not` keyword makes it false. For example, if we create a variable `a` and set it to the value `True` , and then print the value of this variable using `not` , we get the following result:

```
>>> a = True
>>> print(not a)
False
```

And similarly for a `False` value, we get `True` :

```
>>> b = False
>>> print(not b)
True
```

This doesn't seem very useful until you start using the keyword in `if` statements. For example, to find out whether an item is not in a list, we could write something like this:

```
>>> clothing_list = ['shorts', 'undies', 'boxers', 'long johns',
                 'knickers']
>>> if 'pants' not in clothing_list:
        print('You really need to buy some pants')
You really need to buy some pants
```

OR

The `or` keyword joins two conditions in a statement (such as an `if` statement) to say that at least one of the conditions should be true. Here's an example:

```
if dino == 'Tyrannosaurus' or dino == 'Allosaurus':
    print('Carnivores')
elif dino == 'Ankylosaurus' or dino == 'Apatosaurus':
    print('Herbivores')
```

In this case, if the value of the variable `dino` is `Tyrannosaurus` or `Allosaurus` , the program prints `Carnivores` . If it is `Ankylosaurus` or `Apatosaurus` , the program prints `Herbivores` .

PASS

Sometimes when you're developing a program, you want to write only small pieces of it to try things out. The problem with doing this is that you can't have an `if` statement without the block of code that should be run if the condition in the `if` statement is true. You also cannot have a `for` loop without the block of code that should be run in the loop. For example, the following code works just fine:

```
>>> age = 15
>>> if age > 10:
        print('older than 10')
older than 10
```

But if you don't fill in the block of code (the body) for the `if` statement, you'll get an error message:

```
>>> age = 15
>>> if age > 10:
File "<stdin>", line 2
    ^
IndentationError: expected an indented block after 'if' statement on line 1
```

This is the error message Python displays when you should have a block of code after a statement of some kind (it won't even let you type this kind of code if you're using IDLE). In cases like these, you can use the `pass` keyword to write a statement but not provide the block of code that goes with it.

For example, say you want to create a `for` loop with an `if` statement inside it. Perhaps you haven't decided what to put in the `if` statement yet—maybe you'll use the `print` function, or put in a break, or something else. For now, you can use `pass` , and the code will still work (even if it doesn't do exactly what you want yet). Here's our `if` statement again, this time using the `pass` keyword:

```
>>> age = 15
>>> if age > 10:
        pass
```

The following code shows another use of the `pass` keyword:

```
>>> for x in range(0, 7):
>>>     print(f'x is {x}')
>>>     if x == 4:
            pass

x is 0
x is 1
x is 2
x is 3
x is 4
x is 5
x is 6
```

Python still checks whether the `x` variable contains the value `4` every time it executes the block of code in the loop, but it will do nothing as a consequence, so it will print every number in the range 0 to 7. Later, you could add the code in the block for the `if` statement, replacing the `pass` keyword with something else, such as `break` :

```
>>> for x in range(0, 7):
        print(f'x is {x}')
        if x == 4:
            break

x is 1
x is 2
```

```
x is 3
x is 4
```

The `pass` keyword is most commonly used when you're creating a function but don't want to write the code for it yet.

RAISE

The `raise` keyword can be used to cause an error to happen. That might sound like a strange thing to do, but in advanced programming, it can actually be quite useful.

RETURN

The `return` keyword is used to return a value from a function. For example, you might create a function to calculate the number of seconds you've been alive up till your last birthday:

```
def age_in_seconds(age_in_years):
    return age_in_years * 365 * 24 * 60 * 60
```

When you call this function, the returned value can be assigned to another variable or it can be printed:

```
>>> seconds = age_in_seconds(9)
>>> print(seconds)
283824000
>>> print(age_in_seconds(12))
378432000
```

TRY

The `try` keyword begins a block of code that ends with the `except` and `finally` keywords. Together, these `try` / `except` / `finally` blocks of code are used to handle errors in a program, such as making sure that the program displays a useful message to the user rather than an unfriendly Python error. They're very useful in advanced programs.

WHILE

The `while` keyword is a bit like `for` , except that a `for` loop counts through a range (of numbers), but a `while` loop keeps on running while an expression is true. Be careful with `while` loops—if the expression is always true, the loop will never end (this is called an *infinite loop*). Here's an example:

```
>>> x = 1
>>> while x == 1:
        print('hello')
```

If you run this code, it will loop forever, or at least until you close the Python Shell or press CTRL-C to interrupt it. However, the following code will print "hello" nine times (each time adding 1 to the variable `x` , until `x` is no longer less than 10):

```
>>> x = 1
>>> while x < 10:
        print('hello')
        x = x + 1
```

WITH

The `with` keyword is used with a special kind of object to create a block of code, in a similar way to the `try` and `finally` keywords, and then manages resources for that object. This keyword is used in advanced programs.

YIELD

The `yield` keyword is a little bit like `return` , except that it is used with a specific class of object called a *generator* . Generators create values on request, so in that respect, the `range` function behaves like a generator.

B
PYTHON'S BUILT-IN FUNCTIONS

Python has a well-stocked box of programming tools, including a large number of functions and modules that are ready-made for you to use. These built-in tools can make writing programs a lot easier.

As you read in Chapter 7 , modules need to be imported before they can be used. Python's *built-in functions* don't need to be imported first; they're available as soon as the Python Shell starts. In this appendix, we'll look at some of the more useful built-in functions, and then focus on one: the `open` function, which lets you open files to read and write from them.

USING BUILT-IN FUNCTIONS

Let's look at some of the built-in functions that are commonly used by Python programmers. I'll describe what they do and how to use them, and then show examples of how they can help in your programs.

THE ABS FUNCTION

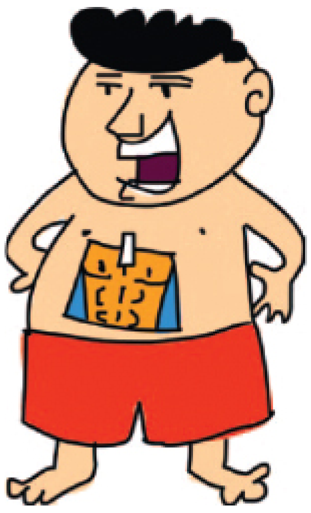
The abs function returns the *absolute value* of a number, which is the value of a number without its sign. For example, the absolute value of 10 is 10, and the absolute value of –10 is 10.

To use the abs function, simply call it with a number or variable as its parameter, like this:

```
>>> print(abs(10))
10
>>> print(abs(-10))
10
```

You might use the abs function to calculate an absolute amount of movement of a character in a game, no matter in which the direction that character is traveling. For example, say the character takes 3 steps to their left (negative 3 or –3) and then 10 steps to their right (positive 10).

If we didn't care about the direction (positive or negative), the absolute value of these numbers would be 3 and 10. You might use this in a board game where you roll two dice and then move your character a maximum number of steps in any direction, based on the total of the dice. Now, if we store the number of steps in a variable, we can determine if the character is moving with the following code. We might want to display some information when the player has decided to move (in this case, we'll display "Character is moving far" or "Character is moving," depending on the number):



```
>>> steps = -3
>>> if abs(steps) > 5:
        print('Character is moving far')
    elif abs(steps) != 0:
        print('Character is moving')
```

If we hadn't used `abs` , the `if` statement might look like this:

```
>>> steps = 10
>>> if steps < -5 or steps > 5:
        print('Character is moving far')
    elif steps != 0:
        print('Character is moving')
```

As you can see, using `abs` made the `if` statement a little shorter and easier to understand.

THE ALL FUNCTION

The `all` function returns `True` if all items in a list (or any other sort of collection) evaluate to `True` . Put most simply, this means that all the values of the items in the list are not 0, `None` , an empty string ('''') or the Boolean value `False` .

So if all the items in the list are nonzero numbers, `all` will return `True` :

```
>>> mylist = [1,2,5,6]
>>> all(mylist)
True
```

But if any of the values are `0` , it will return `False` :

```
>>> mylist = [1, 2, 3, 0]
>>> all(mylist)
False
```

Not just numbers—a mixed list of values that includes `None` will also return `False` :

```
>>> mylist = [100, 'a', None, 'b', True, 'zzz', ' ']
>>> all(mylist)
```

```
False
```

Let’s try that same example again if `None` is removed:

```
>>> mylist = [100, 'a', 'b', True, 'zzz', ' ']
>>> all(mylist)
True
```

THE ANY FUNCTION

The `any` function is similar to `all` , except that if any of the values evaluate to `True` , it will return `True` . Let’s try the same example with the numbers:

```
>>> mylist = [1, 2, 5, 6]
>>> any(mylist)
True
```

Our mixed list of zeros, `None` , empty strings, and `False` also works the same as `all` :

```
>>> mylist = [0, False, None, "", 0, False, '']
>>> any(mylist)
False
```

But if we make a minor change to that list—such as adding a nonzero number like 100—we then get `True` instead:

```
>>> mylist = [0, False, None, "", 0, False, '', 100]
>>> any(mylist)
True
```

THE BIN FUNCTION

The `bin` function converts a number into its *binary representation* . Binary is beyond the scope of this book, but, in short, it’s a numbering system made up of 1s and 0s and is the basis for pretty much everything in computing. Here's a simple example, converting some numbers into binary:

```
>>> bin(100)
'0b1100100'
>>> bin(5)
'0b101'
```

THE BOOL FUNCTION

The name `bool` is short for *Boolean* , the word programmers use to describe a type of data that can have one of two possible values: typically, `True` or `False` .

The `bool` function takes a single parameter and returns either `True` or `False` based on its value. When using `bool` for numbers, `0` returns `False` but any other number returns `True` . Here's how you might use `bool` with various numbers:

```
>>> print(bool(0))
False
>>> print(bool(1))
True
>>> print(bool(1123.23))
True
>>> print(bool(-500))
True
```

When you use `bool` for other values, like strings, it returns `False` if there's no value for the string (in other words, the `None` keyword or an empty string). Otherwise, it will return `True` , as shown here:

```
>>> print(bool(None))
False
>>> print(bool('a'))
True
>>> print(bool(' '))
True
>>> print(bool('What do you call a pig doing karate? Pork Chop!'))
True
```

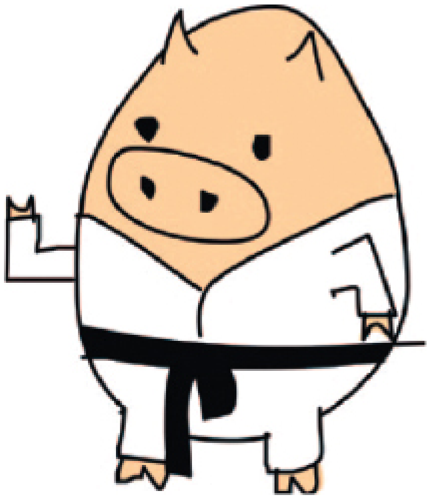
The `bool` function will also return `False` for lists, tuples, and maps that do not contain any values, or `True` when they do:

```
>>> my_silly_list = []
>>> print(bool(my_silly_list))
False
>>> my_silly_list = ['s', 'i', 'l', 'l', 'y']
>>> print(bool(my_silly_list))
True
```

You might use `bool` when you need to decide whether a value has been set or not. For example, if we ask people using our program to enter the year they were born, our `if` statement could use `bool` to test the value they enter:

```
>>> year = input('Year of birth: ')
Year of birth:
>>> if not bool(year):
        print('You need to enter a value for your year of birth')

You need to enter a value for your year of birth
```

The first line of this example uses `input` to store what someone enters on the keyboard as the variable year. Pressing ENTER on the next line (without typing anything else) results in an empty string in the variable. (We also used `sys.stdin.readline()` in Chapter 7 , which is another way to do the same thing.)



On the following line, the `if` statement checks the Boolean value of the variable. Because the user didn't enter anything in this example, the `bool` function returns `False` . The `if` statement uses the `not` keyword, which is a way of saying, "Do this if the function does not return `True` ," and so the code prints `You need to enter a value for your year of birth` on the next line.

THE CALLABLE FUNCTION

The `callable` function simply tells you if something is a function (in other words, can it be called?). The following code returns `False` . . .

```
>>> callable('peas')
False
```

. . . because the string `'peas'` is not a function. But the following code returns `True` . . .

```
>>> callable(bin)
True
```

. . . because `bin` is a function. The following code will also return `True` :

```
>>> class People:
        def run(self):
            print('running')

>>> callable(People.run)
True
```

The `People` class has a single function `run` . If we check whether the class function is callable (it is), we get `True` . Also, if we create an object of the class, and then check whether the object function (`p.run`) is callable, again we get `True` :

```
>>> p = People()
>>> callable(p.run)
True
```

THE CHR FUNCTION

Every character you type in Python has an underlying numeric code identifying it. The character `'a'` for example, has the numeric value 97. A capital `'A'` has the numeric value 65. The `chr` function takes a numeric parameter and returns the character. So we can try the values 97 and 65:

```
>>> chr(97)
'a'
>>> chr(65)
'A'
```

We can try some more random numbers, like 22283, which is a character in the Chinese character set:

```
>>> chr(22283)
'國'
```

Or 949, which is the Greek character *epsilon* :

```
>>> chr(949)
'ε'
```

Or 8595, which isn't a character at all—it's a downward-pointing arrow:

```
>>> chr(8595)
'↓'
```

THE DIR FUNCTION

The `dir` function (short for *directory*) returns information about any value. Basically, it tells you the functions that can be used with that value in alphabetical order.

For example, to display the functions that are available for a list value, enter this:

```
>>> dir(['a', 'short', 'list'])
['__add__', '__class__', '__contains__', '__delattr__',
'__delitem__', '__doc__', '__eq__', '__format__', '__ge__',
'__getattribute__', '__getitem__', '__gt__', '__hash__', '__iadd__',
'__imul__', '__init__', '__iter__', '__le__', '__len__', '__lt__',
'__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__',
'__repr__', '__reversed__', '__rmul__', '__setattr__', '__setitem__',
'__sizeof__', '__str__', '__subclasshook__', 'append', 'count',
'extend', 'index', 'insert', 'pop', 'remove', 'reverse', 'sort']
```

The `dir` function works on pretty much anything, including strings, numbers, functions, modules, objects, and classes. But sometimes the information it returns may not be very useful. For example, if you call `dir` on the number 1, it displays a number of special functions (those that start and end with underscores) used by Python itself, which isn't really useful (you can usually ignore most of them):

```
>>> dir(1)
['__abs__', '__add__', '__and__', '__bool__', '__ceil__', '__class__',
'__delattr__', '__dir__', '__divmod__', '__doc__', '__eq__', '__float__',
'__floor__', '__floordiv__', '__format__', '__ge__', '__getattribute__',
'__getnewargs__', '__gt__', '__hash__', '__index__', '__init__',
'__init_subclass__', '__int__', '__invert__', '__le__', '__lshift__',
'__lt__', '__mod__', '__mul__', '__ne__', '__neg__', '__new__', '__or__',
'__pos__', '__pow__', '__radd__', '__rand__', '__rdivmod__', '__reduce__',
'__reduce_ex__', '__repr__', '__rfloordiv__', '__rlshift__', '__rmod__',
'__rmul__', '__ror__', '__round__', '__rpow__', '__rrshift__', '__rshift__',
'__rsub__', '__rtruediv__', '__rxor__', '__setattr__', '__sizeof__', '__str__',
'__sub__', '__subclasshook__', '__truediv__', '__trunc__', '__xor__',
'as_integer_ratio', 'bit_count', 'bit_length',
'conjugate', 'denominator', 'from_bytes', 'imag', 'numerator',
'real', 'to_bytes']
```

The `dir` function can be useful when you have a variable and want to quickly find out what you can do with it. For example, run `dir` using the variable `popcorn` containing a string value, and you get the list of functions provided by the `string` class (all strings are members of the `string` class):

```
>>> popcorn = 'I love popcorn!'
>>> dir(popcorn)
['__add__', '__class__', '__contains__', '__delattr__', '__dir__', '__doc__',
'__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__',
'__getnewargs__', '__gt__', '__hash__', '__init__', '__init_subclass__',
'__iter__', '__le__', '__len__', '__lt__', '__mod__', '__mul__', '__ne__',
'__new__', '__reduce__', '__reduce_ex__', '__repr__', '__rmod__',
'__rmul__', '__setattr__', '__sizeof__', '__str__',
'__subclasshook__', 'capitalize', 'casefold', 'center', 'count', 'encode',
'endswith', 'expandtabs', 'find', 'format', 'format_map', 'index',
'isalnum', 'isalpha', 'isascii', 'isdecimal', 'isdigit', 'isidentifier',
'islower', 'isnumeric', 'isprintable', 'isspace', 'istitle', 'isupper',
'join', 'ljust', 'lower', 'lstrip', 'maketrans', 'partition',
'removeprefix', 'removesuffix', 'replace', 'rfind', 'rindex', 'rjust',
'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines', 'startswith',
'strip', 'swapcase', 'title', 'translate', 'upper', 'zfill']
```

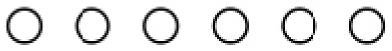
At this point, you could use `help` to get a short description of any function in the list. Here's an example of running `help` for the `upper` function:

```
>>> help(popcorn.upper)
Help on built-in function upper:

upper() method of builtins.str instance
    Return a copy of the string converted to uppercase.
```

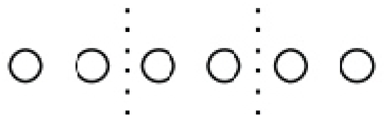
The information returned can be a little confusing, so let's take a closer look. The first line tells you that `upper` is a built-in function of an instance (object) of a string. And the second line tells you exactly what it does (returns a copy of the string in uppercase).

THE DIVMOD FUNCTION

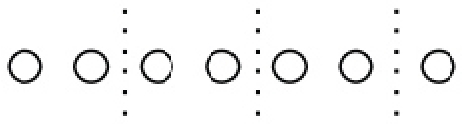
The helper function `divmod` takes two parameters (two numbers representing a dividend and divisor) and then returns the result of dividing the two numbers, along with the result of performing the modulo operation on the two numbers. Division is the mathematical operation of calculating how many times one number can be divided into parts of a second number. For example, how many times can we divide a set of six balls into sets of two balls?



The answer: we can divide it three times.



The `modulo` operation is almost the same thing, except modulo returns the number left over after we do the division. So the result of modulo for the six balls divided by two above is zero (because there's nothing left over). What if we add one more ball? If we divide seven balls into sets of two, the result of the division is still three, but one ball will be left over.



And that's what `divmod` will return as a tuple of two numbers: the result of the division and the result of the modulo operation. Let's try with 6 and 2 first:

```
>>> divmod(6, 2)
(3, 0)
```

And then with 7 and 2:

```
>>> divmod(7, 2)
(3, 1)
```

THE EVAL FUNCTION

The `eval` function (short for *evaluate*) takes a string as a parameter and runs it as though it were a Python expression. For example, `eval('print(''wow'')')` will actually run the statement `print(''wow'')` .

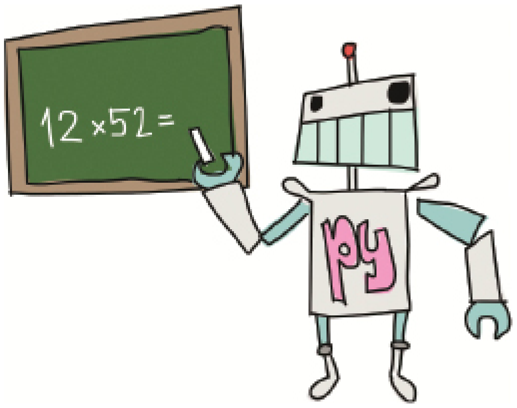
The `eval` function works only with simple expressions, such as the following:

```
>>> eval('10*5')
50
```

Expressions that are split over more than one line (such as `if` statements) generally won't evaluate, as in this example:

```
>>> eval('''if True:
        print("this won't work at all")''')

Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    eval('''if True:
  File "<string>", line 1
    if True:
    ^^
SyntaxError: invalid syntax
```

The `eval` function is often used to turn user input into Python expressions. For example, you could write a simple calculator program that reads equations entered into Python and then calculates (evaluates) the answers.



Because user input is read as a string, Python needs to convert it into numbers and operators before doing any calculations. The `eval` function makes that conversion easy:

```
>>> your_calculation = input('Enter a calculation: ')

Enter a calculation: 12*52
>>> eval(your_calculation)
624
```

In this example, we use the `input` function to read what the user enters into the `your_calculation` variable. On the next line, we enter the expression `12*52` . We use `eval` to run this calculation, and the result is printed on the final line.

THE EXEC FUNCTION

The `exec` function is like `eval` , except you can use it to run more complicated programs. While `eval` returns a value (something you can save in a variable), `exec` does not. Here's an example:

```
>>> my_small_program = '''print('ham')
print('sandwich')'''
>>> exec(my_small_program)
ham
sandwich
```

In the first two lines, we create a variable with a multiline string containing two `print` statements, and then use `exec` to run the string.

You can use `exec` to run mini programs your Python program reads in from files—programs inside programs! This can be quite useful when writing long, complex applications. For example, you could create a *Dueling Robots* game, where two robots move around a screen and try to attack each other. Players of the game would provide the instructions for their robot as mini Python programs. The *Dueling Robots* game would read in these scripts and use `exec` to run.

THE FLOAT FUNCTION

The `float` function converts a string or a number into a *floating-point* number, which is a number with a decimal place (also called a *real number* or *float*). For example, the number 10 is an integer (also called a *whole number*), but 10.0, 10.1, and 10.253 are all floating-point numbers. You might use floating point numbers if you're writing a simple program to calculate monetary amounts. Floats are also used in graphics programs (like 3D games) to calculate how and where to draw things on the screen.

You can convert a string to a float simply by calling `float` :

```
>>> float('12')
12.0
```

You can use a decimal place in a string as well:

```
>>> float('123.456789')
123.456789
```

You might use `float` to convert values entered into your program into proper numbers, which is particularly useful when you need to compare the value a person enters with other values. For example, to check whether a person's age is above a certain number, we could do this:

```
>>> your_age = input('Enter your age: ')

Enter your age: 20
>>> age = float(your_age)
>>> if age > 13:
        print(f'You are {age - 13} years too old')

You are 7.0 years too old
```

THE INPUT FUNCTION

The `input` function is used to read text entered by the person who's using your program—everything they type until they hit the ENTER key. The results are returned in a string for you to use. You can prompt the user of your program to enter something, with a message like this example:

```
>>> s = input('Tell me a play on words:\n')
Tell me a play on words:
A hedgehog went to see a play about a plucky young girl, but left
dis-a-pointed
>>> print(s)
A hedgehog went to see a play about a plucky young girl, but left
dis-a-pointed
```

Or with no message:

```
>>> s = input()
A hedgehog went to see a play about a plucky young girl, but left
dis-a-pointed
```

In either case, the result of the `input` function is the same: a string containing the text. See the previous section on the `float` function for more examples of using the return value.

THE INT FUNCTION

The `int` function converts a string or a number into a whole number (or *integer*), which basically means that everything after the decimal point is dropped. For example, here's how to convert a floating-point number into a plain integer:

```
>>> int(123.456)
123
```

This example converts a string to an integer:

```
>>> int('123')
123
```

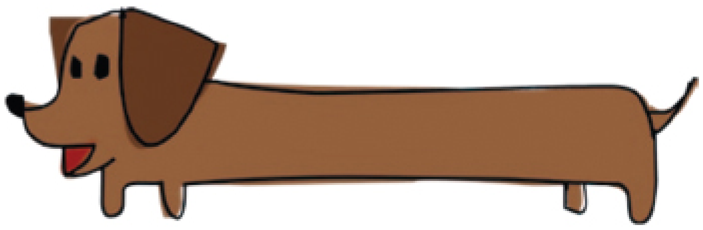
But try to convert a string containing a floating-point number into an integer, and you get an error message. For example, here we try to convert a string containing a floating-point number using the `int` function:

```
>>> int('123.456')
Traceback (most recent call last):
  File "<pyshell>", line 1, in <module>
```

```
    int('123.456')
ValueError: invalid literal for int() with base 10: '123.456'
```

As you can see, the result is a `ValueError` message.

THE LEN FUNCTION



The `len` function returns the length of an object or, in the case of a string, the number of characters in the string. For example, to get the length of `'this is a test string'` , you'd do this:

```
>>> len('this is a test string')
21
```

When used with a list or a tuple, `len` returns the number of items in that list or tuple:

```
>>> creature_list = ['unicorn', 'cyclops', 'fairy', 'elf', 'dragon',
                     'troll']
>>> print(len(creature_list))
6
```

Used with a `dict` (or *dictionary*), `len` also returns the number of items in the `dict` :

```
>>> enemies = {'Batman' : 'Joker',
               'Superman' : 'Lex Luthor',
               'Spiderman' : 'Green Goblin'}
>>> print(len(enemies))
3
```

The `len` function is particularly useful when you're working with loops. For example, we could use it to display the index positions of the elements in a list like so:

```
>>> fruit = ['apple', 'banana', 'clementine', 'dragon fruit']
>>> length = len(fruit)
>>> for x in range(0, length):
        print(f'the fruit at index {x} is {fruit[x]}')

the fruit at index 0 is apple
the fruit at index 1 is banana
the fruit at index 2 is clementine
the fruit at index 3 is dragon fruit
```

First, we store the length of the list in the variable `length` , and then use that variable in the `range` function to create our loop. As we loop through each item in the list, we print a message showing the item's index position and value. You could also use the `len` function, if you had a list of strings and wanted to print every second or third item in the list.

THE LIST FUNCTION

If you call `list` without any parameters, you'll get an empty list object in response. There's no difference between `list()` and using square brackets at that point. We can check if this is actually the case by testing if the two lists are equal (`==`):

```
>>> l1 = list()
>>> l2 = []
>>> l1 == l2
True
```

While this might not seem particularly useful, `list` can also be used to convert certain types of Python objects (called *iterables*) into a list. The simplest example of this would be to use the `range` function (this function is described on page 317) with `list` :

```
>>> list(range(0, 10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

THE MAX AND MIN FUNCTIONS

The `max` function returns the largest item in a list, tuple, or string. For example, here's how to use it with a list of numbers:

```
>>> numbers = [5, 4, 10, 30, 22]
>>> print(max(numbers))
30
```

You can do exactly the same thing with a string, or a list of strings:

```
>>> strings = 'stringSTRING'
>>> print(max(strings))
t
>>> strings = ['s', 't', 'r', 'i', 'n', 'g', 'S', 'T', 'R', 'I', 'N', 'G']
>>> print(max(strings))
t
```

Letters are ranked alphabetically, but lowercase letters come after uppercase letters, so `t` is more than `T`. But you don't have to use lists, tuples, or strings. You can also call the max function directly, and enter the items that you want to compare into the parentheses as parameters:

```
>>> print(max(10, 300, 450, 50, 90))
450
```

The `min` function works like `max` , except it returns the smallest item in the list, tuple, or string. Here's our list of numbers example using `min` instead of `max` :

```
>>> numbers = [5, 4, 10, 30, 22]
>>> print(min(numbers))
4
```

Suppose you're playing a guessing game with a team of four players, and each has to guess a number that is less than your number. If any player guesses above your number, all players lose, but if they all guess lower, they win. We could use `max` to quickly find whether any of the guesses are higher, like so:

```
>>> guess_this_number = 61
>>> player_guesses = [12, 15, 70, 45]
>>> if max(player_guesses) > guess_this_number:
        print('Boom! You all lose')
    else:
        print('You win')

Boom! You all lose
```

In this example, we store the number to guess using the variable `guess_this_number` . The team members' guesses are stored in the list `player_guesses` . The `if` statement checks the maximum guess against the number in `guess_this_number` , and if any player guesses over the number, we print the message "Boom! You all lose."

THE ORD FUNCTION

The `ord` function is basically the reverse of the `chr` function: whereas `chr` converts a number to a character, `ord` tells you what the number code is for a character. Here are some examples:

```
>>> ord('a')
97
>>> ord('A')
65
```

```
>>> ord('國')
22283
```

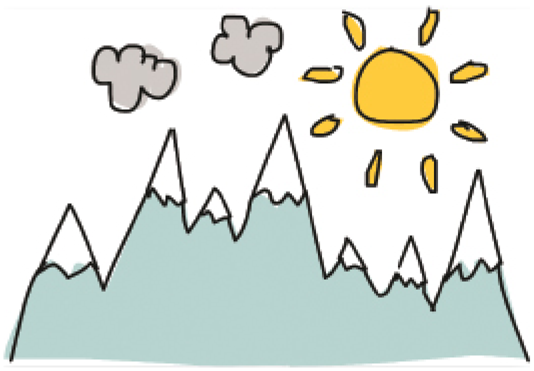
THE POW FUNCTION

The `pow` function takes two numbers and calculates the value of one number (let's call this `x`) to the power of the other number (let's call this `y`). Essentially, `pow` will multiply `x` by itself `y` times. For example, 2 to the power of 3 (in mathematics terms this is 2^3) would be 2 * 2 * 2 (or in mathematical notation, $2 \times 2 \times 2$), which is 8 (2 * 2 is 4, 4 * 2 is 8). Another example: 3 to the power of 3 (3^3) is 27. Let's see how this looks in code:

```
>>> pow(2, 3)
8
>>> pow(3, 3)
27
```

THE RANGE FUNCTION

The `range` function is mainly used in `for` loops to loop through a section of code a specific number of times. The first two parameters given to `range` are called the *start* and the *stop* . You've seen `range` with these two parameters in the earlier example using the `len` function to work with a loop.



The numbers that `range` generates begin with the number given as the first parameter and end with the number that's one less than the second parameter. For example, the following shows what happens when we print the numbers that `range` creates between 0 and 5:

```
>>> for x in range(0, 5):
        print(x)

0
1
2
3
4
```

The `range` function actually returns a special object called an *iterator* that repeats an action a number of times. In this case, it returns the next highest number each time it is called.

You can convert the iterator into a list by using the `list` function. If you then print the returned value when calling `range` , you'll see the numbers it contains as well:

```
>>> print(list(range(0, 5)))
[0, 1, 2, 3, 4]
```

You can also add a third parameter to `range` , called `step` . If the `step` value is not included, the number 1 is used as the `step` by default. But what happens when we pass in the number 2 as the `step` ? Here's the result:

```
>>> count_by_twos = list(range(0, 30, 2))
>>> print(count_by_twos)
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28]
```

Each number in the list increases by two from the previous number, and the list ends with the number 28, which is 2 less than 30. You can also use negative steps:

```
>>> count_down_by_twos = list(range(40, 10, -2))
>>> print(count_down_by_twos)
[40, 38, 36, 34, 32, 30, 28, 26, 24, 22, 20, 18, 16, 14, 12]
```

THE SUM FUNCTION

The `sum` function adds up items in a list and returns the total. Here's an example:

```
>>> my_list_of_numbers = list(range(0, 500, 50))
>>> print(my_list_of_numbers)
[0, 50, 100, 150, 200, 250, 300, 350, 400, 450]
>>> print(sum(my_list_of_numbers))
2250
```

On the first line, we create a list of numbers between 0 and 500, using `range` with a step of 50. Next, we print the list to see the result. Lastly, passing the variable `my_list_of_numbers` to the `sum` function with `print(sum(my_list_of_numbers))` adds all the items in the list, giving the total of 2250.

OPENING A FILE IN PYTHON

Python's built-in `open` function opens a file, so you can do something useful with it (like display the contents). How you tell the function which file to open depends on your operating system. Read over the example for a Windows file, and then read the Mac or Ubuntu-specific section if you're using one of those systems. First create a plaintext file in your home folder called *test.txt* —on Windows, you can use Notepad; on Ubuntu Linux or Raspberry Pi, use TextEditor; on macOS, use TextEdit (but in TextEdit, you will need to select **Format ▸ Make Plain Text**). You can put whatever you like in the file.

OPENING A WINDOWS FILE

If you're using Windows, enter the following code to open *test.txt* :

```
>>> test_file = open('c:\\Users\\<your username>\\test.txt')
>>> text = test_file.read()
>>> print(text)
There once was a boy named Marcelo
Who dreamed he ate a marshmallow
He awoke with a start
```

```
As his bed fell apart
And he found he was a much rounder fellow
```

On the first line, we use `open` , which returns a file object with functions for working with files. The parameter we use with the `open` function is a string telling Python where to find the file. If you're using Windows, you saved *test.txt* to your user folder on the *C:* drive, so you specify the location of your file as `c:\Users\<your username>\test.txt` . (Don't forget to replace `<your username>` with your actual username!)

The two backslashes in the Windows filename tell Python that the backslash is just that, and not some sort of command. (As you read in Chapter 3 , backslashes on their own have a special meaning in Python, particularly in strings.) We save the file object to the `test_file` variable.

On the second line, we use the `read` function, provided by the file object, to read the contents of the file and store it in the `text` variable. We print the variable on the final line to display the contents of the file.

OPENING A MACOS FILE

If you are using macOS, you'll need to enter a different location on the first line of the Windows example to open *test.txt* . Use the username you clicked when saving the text file in the string. For example, if the username is *sarahwinters* , the `open` parameter should look like this:

```
>>> test_file = open('/Users/sarahwinters/test.txt')
```

OPENING AN UBUNTU OR RASPBERRY PI FILE

If you are using Ubuntu Linux or Raspberry Pi, you'll need to enter a different location on the first line of the Windows example to open *test.txt* . Use the username you clicked when saving the text file. For example, if the username is *jacob* , the `open` parameter should look like this:

```
>>> test_file = open('/home/jacob/test.txt')
```

WRITING TO FILES

The file object returned by `open` has other functions besides `read` . We can create a new, empty file by using a second parameter—the string `'w'` — when we call `open` (this parameter tells Python that we want to write to the file object, rather than read from it):

```
>>> test_file = open('c:\\Users\\rachel\\myfile.txt', 'w')
```

We can now add information to this new file using the `write` function:

```
>>> test_file = open('c:\\Users\\rachel\\myfile.txt', 'w')
>>> test_file.write('What is green and loud? A froghorn!')
20
```

Finally, we need to tell Python when we're finished writing to the file, using the `close` function:

```
>>> test_file = open('c:\\Users\\rachel\\myfile.txt', 'w')
>>> test_file.write('What is green and loud? A froghorn!')
>>> test_file.close()
```

Now, if you open the file with your text editor, you should see that it contains the text "What is green and loud? A froghorn!" Or, you can use Python to read it again:

```
>>> test_file = open('c:\\Users\\rachel\\myfile.txt')
>>> print(test_file.read())
What is green and loud? A froghorn!
```

C
TROUBLESHOOTING

In this appendix, you'll find information on how to fix some less common problems with Python. If you happen to be running older versions of some operating systems, you may experience these issues.

"TK" ERRORS IMPORTING TURTLE ON UBUNTU

If you're using an older version of Ubuntu Linux and get errors when you import turtle, you might need to install a piece of software called *tkinter* . To do so, open the Ubuntu Software Center and enter **python-tk** in the search box. "Tkinter—Writing Tk Applications with Python" should appear in the window. Click **Install** to install this package. This shouldn't be required if you're running a more recent version of Ubuntu—if possible, you should get the owner of your computer to update it for you.

ATTRIBUTE ERROR USING TURTLE

Some new programmers experience strange *attribute* errors when trying to use turtle:

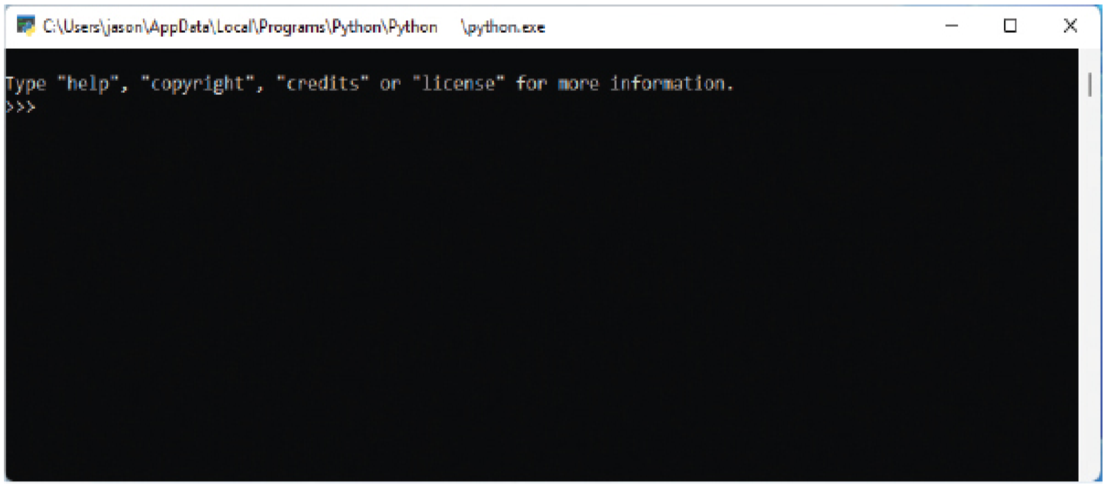
```
>>> import turtle
>>> t = turtle.Turtle()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: module 'turtle' has no attribute 'Turtle'
```

The most typical cause of this error is when you have created a file called *turtle.py* in your home folder. In this case, when you enter `import turtle` , you're getting the file you created and not Python's `turtle` module. If you delete or rename that file, the correct module should import correctly.

PROBLEMS RUNNING TURTLE

If you have problems when using the `turtle` module, and the turtle window itself doesn't appear to be working, try using the Python console instead of the Python Shell, as follows:

- In Windows, enter **Python** in the search box and click **Python 3.1 *x*** in the Apps list. You can use the Windows command prompt instead (click the Windows icon and type **cmd** in the search box). When that opens, you'll need to enter the path to a program called *python.exe* . If you've installed Python 3.10, the path might be something like this: *AppData\Local\Programs\Python \Python310\python.exe* . However, it very much depends on what version of Python you've installed, so this method should probably be the last resort (you can see the result of running this in Figure C-1).


C:\Users\jason\AppData\Local\Programs\Python\Python \python.exe

```
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Figure C-1: Running the Python console from Windows command prompt

- In macOS, click the Spotlight Search icon at the top-right corner of the screen (it should look like a magnifying glass) and enter **Terminal** in the input box. Then enter `python3` when the terminal opens.
- In Ubuntu Linux, open the terminal from your **Show Applications** menu and enter `python3.10` (note that your version number may be different).
- In Raspberry Pi, click the Terminal icon on the menu bar at the top, or click **Terminal** in the **Accessories** menu and enter `/usr/local/opt/python-3.10.0` (this will work only if you followed the Raspberry Pi installation instructions in Chapter 1 ; note that your version number may be different).

The Python console is similar to the Python Shell (IDLE), but it doesn't have syntax highlighting (colored text), easy save options, and other beneficial features. However, if you're having problems running turtle in the Python Shell, using the Python console might help.

CLASS TAKES NO ARGUMENTS

A common error some readers hit is a `TypeError` , usually first seen in Chapter 11 . You might see an error similar to the following:

```
b = Ball(canvas, 'red')
Traceback (most recent call last):
  File "/usr/lib/python3.10/idlelib/run.py", line 573, in runcode
    exec(code, self.locals)
  File "<pyshell#4>", line 1, in <module>
TypeError: Ball() takes no arguments
```

The reason for this is generally missing underscores. The `Ball` class is first defined like this:

```
class Ball:
    def __init__(self, canvas, color):
        self.canvas = canvas
        self.id = canvas.create_oval(10, 10, 25, 25, fill=color)
        self.canvas.move(self.id, 245, 100)
```

However, if you mistype the `__init__` function with a single underscore on either side (`_init_`), Python will no longer recognize it as an initialization function. This is why calling `Ball(...)` with any arguments results in an error—Python thinks there is no initialization function to call (in fact, it creates a default initialization function for you that has no parameters).

INDEX

SYMBOLS AND NUMBERS

UPDATES

Visit *http://python-for-kids.com* for updates, errata, and other information.

COLOPHON

The fonts used in *Python for Kids, 2nd Edition, Second Edition* are New Baskerville, Futura, The Sans Mono Condensed and Dogma. The book was typeset with LATEX 2 $_\varepsilon$ package `nostarch` by Boris Veytsman *(2008/06/06 v1.3 Typesetting books for No Starch Press).*