CHAPMAN & HALL/CRC
TEXTBOOKS IN COMPUTING

A CONCISE INTRODUCTION TO PROGRAMMING IN PYTHON

Mark J. Johnson

CRC Press
Taylor & Francis Group
A CHAPMAN & HALL BOOK

A CONCISE INTRODUCTION TO PROGRAMMING IN PYTHON

CHAPMAN & HALL/CRC TEXTBOOKS IN COMPUTING

Series Editors

John Impagliazzo
Professor Emeritus, Hofstra University

Andrew McGettrick
Department of Computer and Information Sciences
University of Strathclyde

Aims and Scope

This series covers traditional areas of computing, as well as related technical areas, such as software engineering, artificial intelligence, computer engineering, information systems, and information technology. The series will accommodate textbooks for undergraduate and graduate students, generally adhering to worldwide curriculum standards from professional societies. The editors wish to encourage new and imaginative ideas and proposals, and are keen to help and encourage new authors. The editors welcome proposals that: provide groundbreaking and imaginative perspectives on aspects of computing; present topics in a new and exciting context; open up opportunities for emerging areas, such as multi-media, security, and mobile systems; capture new developments and applications in emerging fields of computing; and address topics that provide support for computing, such as mathematics, statistics, life and physical sciences, and business.

Published Titles

Pascal Hitzler, Markus Krötzsch, and Sebastian Rudolph, Foundations of Semantic Web Technologies

Uvais Qidwai and C.H. Chen, Digital Image Processing: An Algorithmic Approach with MATLAB®

Henrik Bærbak Christensen, Flexible, Reliable Software: Using Patterns and Agile Development

John S. Conery, Explorations in Computing: An Introduction to Computer Science

Lisa C. Kaczmarczyk, Computers and Society: Computing for Good

Mark J. Johnson, A Concise Introduction to Programming in Python

CHAPMAN & HALL/CRC
TEXTBOOKS IN COMPUTING

A CONCISE INTRODUCTION TO PROGRAMMING IN PYTHON

Mark J. Johnson

CRC Press
Taylor & Francis Group
Boca Raton London New York

CRC Press is an imprint of the
Taylor & Francis Group, an **informa** business

A CHAPMAN & HALL BOOK

CRC Press
Taylor & Francis Group
6000 Broken Sound Parkway NW, Suite 300
Boca Raton, FL 33487-2742

Version Date: 20111110

International Standard Book Number-13: 978-1-4398-9695-2 (eBook - PDF)

Contents

Code Listings

This page intentionally left blank

Preface

Welcome!

This text provides an introduction to writing software in Python. No previous programming experience is necessary.

Most chapters begin with an example program illustrating a small set of new concepts. These programs are available for download at `http://www.central.edu/go/conciseintro/`. However, you might consider typing each program in by hand rather than using the downloaded files. The reason is that you can learn by typing, both in terms of thoroughly reading the programs, and responding to error messages that result from typos. Rather than being a hindrance, learning to deal with error messages will be quite helpful as you write your own programs.

To Instructors

This text is designed for a first course in computer science and is suitable for majors and non-majors. Among its features:

- Chapters are short, intended for one or at most two class periods. This provides a flexible framework to build a course around.
- Explanations are brief and precise.
- Basic procedural constructs such as functions, selection, and repetition are introduced early, allowing them to be used throughout the semester.
- Objects are (explicitly) used in the middle of the course, and writing classes comes toward the end.
- Examples, exercises, and projects are from a wide range of application domains, including biology, physics, images, sound, mathematics, games, and textual analysis.
- No external libraries are required. All example programs run in standard Python 3.

This text is designed to help teach programming rather than being an encyclopedic reference. Topics are introduced as needed for the examples, and the focus is always on what a beginning student might need to know at that point.

The first few exercises in each chapter are intended to reinforce comprehension, but beyond that, all of the other exercises ask students to write code. Some chapters may seem more like interludes than full chapters, with no example program and relatively few exercises. This variation is intentional: it allows time for students to work on exercises or projects from earlier chapters, while still moving forward with some new material.

Parts I and II contain the core of an introductory course and are designed to be worked through in order. The topics in Part III are grouped as follows:

	Chapters
Sound	19, 20
Images	21, 22
Class Design	23, 24, 25
Functional Programming	26
Parallel Programming	27
GUI Design	28

The main dependency across these groups is that the GUI example is written using inheritance from Chapter 25.

Images are a bit tricky to work with in Python 3. At this time, there is no standard Python image module, and the Python Imaging Library (PIL) has not been updated to support Python 3. Once it is compatible with Python 3, PIL will likely still require a separate installation that involves other external libraries. Thus, the image chapters use a custom class that reads and writes simple ASCII PPM files. This format allows students to open images in a text editor and to analyze the custom class itself as a case study later in the course. You may want to install GIMP on lab computers to facilitate converting and viewing PPM files.

Check the book's website at `http://www.central.edu/go/conciseintro/` for sample files, errata, and other resources.

Feedback

Feel free to contact me at `johnsonm@central.edu`. I would appreciate hearing any comments, suggestions, or corrections you might have.

Acknowledgments

Many thanks to my colleagues Stephen Fyfe and Robert Franks for their conversations, suggestions, and support, and to Central College for the sabbatical that allowed this project to grow from a set of notes into a full text. Thanks also to the manuscript reviewers who provided very helpful comments and suggestions.

About the Author

Mark J. Johnson is professor of computer science and mathematics at Central College in Pella, Iowa, where he holds the Ruth and Marvin Denekas Endowed Chair in Science and Humanities. Mark is a graduate of the University of Wisconsin-Madison (Ph.D., mathematics) and St. Olaf College.

This page intentionally left blank

Part I

Foundations

This page intentionally left blank

Chapter 1

Computer Systems and Software

Programmable software is what makes a computer a powerful tool. Each different program essentially "rewires" the computer to allow it to perform a different task. In this course, you will learn basic principles of writing software in the Python programming language.

Python is a popular scripting language available as a free download from `http://www.python.org/`. Follow the instructions given there to install the latest production version of Python 3 on your system. The examples in this text were written with Python 3.2.

The CPU and RAM

In order to write software, it will be helpful to be able to imagine what happens inside a computer when your program runs. We begin with a rough picture and gradually fill in details along the way.

When a program is ready to run, it is loaded into RAM, usually from long-term storage such as a hard drive. **RAM** is an acronym for random access memory, which is the working **memory** of a computer. RAM is **volatile**, meaning that it requires electricity to keep its contents.

Once a program is loaded into RAM, the **CPU**, or central processing unit, executes the instructions of the program, one at a time. Each type of CPU has its own **instruction set**, and you might be surprised at how limited these instruction sets are. Most instructions boil down to a few simple types: load data, perform arithmetic, and store data. What is amazing is that these small steps can be combined in different ways to build programs that are enormously complex, such as games, spreadsheets, and physics simulations.

Computer Languages

CPU instruction sets are also known as **machine languages**. The key point to remember about machine languages is that in order to be run by a CPU, a program *must* be written in the machine language of that CPU. Unfortunately, machine languages are not meant to be read or written by humans. They are really just specific sequences of bits in memory. (We will explain bits later if you do not know what they are.)

Because of this, people usually write software in a **higher-level language**:

Level	Language	Purposes
Higher	Python	Scripts
	Java	Applications
	C, C++	Applications, Systems
	Assembly Languages	Specialized Tasks
Lower	Machine Languages	

This table is not meant to be precise, but, for example, most programmers would agree that C and C++ are closer to the machine than Python.

Compilation and Interpretation

Now if CPUs can only run programs written in their own machine language, how do we run programs written in Python, Java, or C++? The answer is that the program must be translated into machine language first.

There are two main types of translation: compilation and interpretation. When a program is **compiled**, it is completely translated into machine language, producing an executable file. C and C++ programs are usually compiled, and when they are compiled in Microsoft Windows®, for example, the executable files generally end in `.exe`.

On the other hand, when a program is **interpreted**, it is translated "on-the-fly." No separate executable file is created. Instead, the translator program (the **interpreter**) is running and it translates your program so that the CPU can execute it. Python programs are usually interpreted.

The Python Interpreter

When you start Python, you are in immediate contact with a Python interpreter. If you provide it with legitimate Python, the interpreter will translate your code so that it can be executed by the CPU. The interpreter displays the version of Python that it will interpret and then shows that it is ready for your input with this prompt:

```
>>>
```

The interpreter will translate and execute any legal Python code that is typed at the prompt. For example, if we enter the following statement:

```
>>> print("Hello!")
```

the interpreter will respond accordingly. Try it and see.

Remember that as you learn new Python constructs, you can always try them out in the interpreter without having to write a complete program. Experiment—the interpreter will not mind.

A Python Program

Still, our focus will be on writing complete Python programs. Here is a short example:

Listing 1.1: Area of a Circle

```
from math import pi
r = 12
area = pi * r ** 2
print("The area of a circle with radius", r, "is", area)
```

Even if you have never seen Python before, you can probably figure out what this program does.

Start the Python **IDLE** application and choose "New Window" from the File menu. Type Listing 1.1 into the new window that appears. Save it as `circle.py`, and then either choose "Run Module" from the Run menu or press F5 to run the program. This program illustrates many important Python concepts, including variables, numeric expressions, assignment, output, and using the standard library. We will examine these components in the next chapter.

Why Computer Science?

Here are a few things to consider as we begin:

1. Software is everywhere. If you are skeptical, search for "weather forecast toaster."

2. Similarly, computation is having a significant impact on the way other disciplines do their work. And for almost every field X, there is a new interdisciplinary field "Computational X."

3. Programming develops your ability to solve problems. Because machine languages are so simplistic, you have to tell the computer *everything* it needs to do in order to solve a problem. Furthermore, running a program provides concrete feedback on whether or not your solution is correct.

4. Computer science develops your ability to understand systems. Software systems are among the most complicated artifacts ever created by humans, and learning to manage complexity in a program will help you learn to manage it in other areas.

5. Programming languages are tools for creation: they let you build cool things. There is nothing quite like getting an idea for a program and seeing it come to life. And then showing it to all your friends.

Exercises

1.1 Experiment with Listing 1.1 by making changes to various parts of the code. Be sure to try some things that break the program (cause errors), just to see how the interpreter reacts.

1.2 Modify Listing 1.1 to also compute and display the circumference of the circle.

1.3 The December 1978 issue of the IEEE Computer Society journal *Computer* contained the following description of a new computer that fit "on the top of any business desk":

> The PCC 2000 consists of a 3MHz 8085A microprocessor, two 32K memory boards, two FD514 double-density, 8.5-inch floppy disk drives, a 12-inch upper/lower case video display...

(a) Compare the PCC 2000's CPU speed and amount of RAM to current desktops.

(b) Does the PCC 2000 appear to have a hard disk? If not, what does it use for long-term storage?

(c) Research the capacity of one of these floppy disks.

(d) List possible reasons that the PCC 2000 has two floppy disk drives instead of one.

Chapter 2

Python Program Components

Let's look at another Python program that creates short nonsense sentences:

Listing 2.1: Mad Lib

```
# madlib.py
# Your Name

adjective = input("Enter an adjective: ")
noun = input("Enter a noun: ")
verb = input("Enter a verb: ")
adverb = input("Enter an adverb: ")
print("A", adjective, noun, "should never", verb, adverb)
```

Type this program into your Python environment, save it as `madlib.py`, and run it a few times to play with it. Enter any responses you like when the interpreter asks for them. Then consider the following aspects of both this program and Listing 1.1.

Variables

Every program accomplishes its work by manipulating data. **Variables** are names that refer to data in memory. Variable names, also known as **identifiers** in Python, may consist of upper and lower alphabetic characters, the underscore (_), and, except for the first character, the digits 0–9. There is a small set of reserved Python **keywords** that may not be used as variable names; otherwise, you are free to choose any names you wish. You should already be able to see how meaningful identifiers can make a program easier to follow. The variables in Listing 2.1 are `adjective`, `noun`, `verb`, and `adverb`.

Program Statements and Syntax

A program is a sequence of **statements**, which are individual commands executed one after another by the Python interpreter. Every statement must have the correct syntax; the **syntax** of any language is the precise form that it is written in. Thus, if a statement does not have the correct form, the Python

interpreter will respond with a **syntax error**. Three types of statements are used in Listings 1.1 and 2.1:

Assignment statements are used to give variables a value. The syntax of an assignment statement is:

```
<variable> = <expression>
```

Read assignment statements from right to left:

1. Evaluate the expression on the right.
2. Assign the variable on the left to refer to that value.

For example, the assignment

```
x = 10 - 17 + 5
```

computes the right side to be -2 and then assigns `x` to refer to -2.

$\Longrightarrow$ Caution: An assignment statement "`=`" is not like a mathematical equals sign. Technically, it is a **delimiter** because it helps the interpreter delimit between the variable on the left and the expression on the right.

Print statements are used to produce program output. The syntax of a `print()` statement is:

```
print(<expression1>, <expression2>, ...)
```

Expressions inside quotation marks (either single or double) are printed literally, whereas expressions outside quotation marks have their value printed. Expressions are separated by single spaces in the output, and each print statement produces output on a separate line.

Technically, `print()` is a **built-in function** that we call in order to print, but it is generally called as its own separate statement.

Import statements are used to access library functions or data, as in the first line of Listing 1.1. The **Python Standard Library** consists of many **modules**, each of which adds a specific set of additional functionality. The statement:

```
from <module> import <name1>, <name2>, ...
```

allows the listed names to be used in your program.

The `math` module includes the constant `pi` and functions such as `sin`, `cos`, `log` and `exp`. Both **`import`** and **`from`** are reserved Python keywords and so may not be used as the names of variables. Keywords appear in bold in program listings.

Data Types

Listings 1.1 and 2.1 use three different types of data:

Strings are sequences of characters inside single or double quotation marks.

Integers are whole number values such as 847, −19, or 7.

Floats (short for "floating point") are values that use a decimal point and therefore may have a fractional part. For example, 3.14159, −23.8, and even 7.0 (because it has a decimal point) are all considered floats.

A variable in Python may refer to any type of data.

Expressions

Recall that the right-hand side of every assignment statement must be an **expression**, which in Python is just something that can be evaluated to give a value.

Input expressions are used to ask the user of a program for information. They almost always appear on the right side of an assignment statement; using an `input` expression allows the variable on the left side to refer to different values each time the program is run.

`input(prompt)`	Display `prompt` and return user input as a string.

The **prompt** is printed to alert the user that the program is waiting for input.

Numeric expressions use the **arithmetic operations** `+`, `-`, `*`, `/`, `//`, and `**` for addition, subtraction, multiplication, division, integer division, and raising to a power. These follow the normal rules of arithmetic and **operator precedence**: exponentiation is done first, then multiplication and division (left to right), and finally addition and subtraction (also left to right). Thus, `1 + 2 * 3` evaluates to `7` because multiplication is done before addition. Parentheses may be used to change the order of operations.

Integer division rounds down to the nearest integer, so `7//2` equals 3 and `-1//3` equals −1. Integer division applied to floats rounds down to the nearest integer, but the result is still of type float.

Finally, variables must be assigned a value before being used in an expression.

Comments

Comments begin with a pound sign `#` and signify that whatever follows is to be completely ignored by the interpreter.

```
# Text that helps explain your program to others
```

Comments may appear anywhere in a Python program and are meant for human readers rather than the interpreter, in order to explain some aspect of the code.

Recap

There was a lot of terminology here, so let's summarize what is most important:

1. Variables refer to data...
2. ... via assignment statements. Remember to read them from right to left.
3. Data can be of type string, integer, or float (so far). The data type of a variable determines what you can do with it.

Exercises

2.1 Give three names that are not legal Python identifiers.

2.2 Use Listing 1.1 to:

(a) List each variable and give the type of data stored in it. Hint: `pi` is a variable.

(b) Identify the assignment statements, and explain the effect of each.

2.3 Determine the output of Listing 1.1 if the `print()` statement is changed to:

```
print("The area of a circle with radius r is area")
```

Explain the result.

2.4 Use Listing 2.1 to:

(a) List each assignment statement and identify the variable whose content changes in each.

(b) Describe what happens if the final space in any of the `input` statements is deleted.

2.5 Determine the value of each of these Python expressions:

(a) `1 + 2 * 3 - 4 * 5 + 6`

(b) `1 + 2 ** 3 * 4 - 5`

(c) `1 / 2 - 3 / 4`

(d) `1 // 2 - 3 // 4 + 5 // 6`

(e) `1 + 4 - 2 / 2`

(f) `(1 + 4 - 2) / 2`

2.6 Determine the output of each of these code fragments:

(a)
```
x = 10
y = 15
print(x, y)
y = x
print(x, y)
x = 5
print(x, y)
```

(b)
```
x = 10
y = 15
print(x, y)
y = x
x = y
print(x, y)
x = 5
print(x, y)
```

(c)
```
x = 10
y = 15
z = 20
x = z
z = y
y = x
print(x, y, z)
```

2.7 Modify Listing 2.1 to create your own Mad Lib program. You may want to look ahead to Chapter 8 to use the `sep=` or `end=` options of the `print()` statement to better control your output.

2.8 Explain the difficulty with using `input()` to get numeric values at this stage. (We will address it soon.)

2.9 Modify Listing 1.1 to write a program `average.py` that calculates and prints the average of two numbers.

2.10 Modify Listing 1.1 to write a program `rect.py` that calculates and prints the area and perimeter of a rectangle given its length and width. Run your program with several different values of the length and width to test it.

2.11 Modify Listing 1.1 to write a program `cube.py` that calculates and prints the volume and surface area of a cube given its width. Run your program with different values of the width to find the point at which volume equals surface area.

2.12 Modify Listing 1.1 to write a program `sphere.py` that calculates and prints the volume and surface area of a sphere given its radius. Look up the formulae if you do not remember them. Run your program with different values of the radius to find the point at which volume equals surface area.

Chapter 3

Functions

Functions in Python allow you to break a task down into appropriate subtasks. They are one of the powerful tools of abstraction that higher-level programming languages provide. Ideally, every function has a single, well-defined purpose.

Consider this example, which uses the `sqrt()` function from the `math` module to implement `hypot()`, also in the `math` module.

Listing 3.1: Hypotenuse

```
1  # hypot.py
2
3  from math import sqrt
4
5  def myhypot(x, y):
6      return sqrt(x ** 2 + y ** 2)
7
8  def main():
9      a = float(input("a: "))
10     b = float(input("b: "))
11     print("Hypotenuse:", myhypot(a, b))
12
13 main()
```

Each chapter, as we begin with a new example, type the program in to your programming environment (such as IDLE), save it with the indicated name, and run it a few times before you continue. Try to determine how the program works, even though it may use new features that you have not seen before.

For our purposes, a Python **program** can be thought of as a collection of function definitions with one main function call. Listing 3.1 defines and calls two functions: `myhypot()` and `main()`. A function cannot be *called* unless it has been *defined*. The function `myhypot()` is defined in lines 5–6 and called in line 11. The function `main()` is defined in lines 8–11 and called in line 13. The order of function definitions is not important: as long as `myhypot()` is defined somewhere in this file, the definition of `main()` could have come first.

Code that is outside all function definitions (like all of Listings 1.1 and 2.1) is directly executed when the program is run. In Listing 3.1, only the function call in line 13 is outside all function definitions, and so it will be executed when the program is run. In a sense, it "drives" execution of the whole program. Because of this, the `main()` function is known as the **driver**. Usually, the driver definition and call are put at the end of a Python program.

Defining a Function

A function **definition** specifies the code that will be executed when the function is called. The syntax of a function definition looks like this:

```
def <function>(<parameters>):
    <body>
```

The **`def`** line must end with a **colon**. The colon signals the beginning of an indented **block** of code, in this case called the **body** of the function. The body is the code that will be executed when the function is called and may consist of any number of lines. In Listing 3.1, line 6 is the entire body of the `myhypot()` function.

Parameters are used to send additional information to a function so that it can do its job. The `<parameters>` in a function definition are optional. In Listing 3.1, the `myhypot()` function has two parameters named `x` and `y`.

Usually, function definitions appear near the top of Python programs, after any **`import`** statements, but before any directly executable code.

Calling a Function

Once a function has been defined, it may be **called**, meaning that it will be executed with particular **arguments** passed as the values for its parameters. The syntax for a function call is:

```
<function>(<arguments>)
```

When this expression appears in a program statement that is being executed, the function body executes, using the argument values as the values of the parameters.

In Listing 3.1, the function `myhypot()` is called in line 11. The arguments being **passed** or sent to `myhypot()` are the values of `a` and `b` at the time the program runs, after the `input()` expressions. The value of `a` will be used as the value of `x` in `myhypot()`, and the value of `b` will be used as `y`. For example, if you enter 4 and 7 in response to the `input()` expressions, then `x` will get the value 4 (the value of `a`) and `y` will get 7 (the value of `b`) when `myhypot()` is called on line 11.

Note: the name of an argument does *not* have to be the same as its corresponding parameter.

Function calls may be **nested** as in lines 9 and 10 of Listing 3.1. The inner function (in these cases, `input()`) runs first, and then the output of that function is immediately sent as the argument to the outer function (`float()`).

Return Statements

A **`return`** statement may optionally appear anywhere in the body of a function, and looks like this:

```
return <expression>
```

When this statement is executed inside of a function, it immediately terminates the function and returns the value of its `<expression>` as the value of the function call. In fact, the `<expression>` is also optional: if no expression is given, the value returned is the special value `None`.

In Listing 3.1, the **`return`** statement in line 6 computes and returns the length of the hypotenuse. The two steps of computing and returning could be separated:

```
def myhypot(x, y):
    hyp = sqrt(x ** 2 + y ** 2)
    return hyp
```

This version will work exactly the same as the original.

If a function does not contain a **`return`** statement, the function returns the value `None` when the last executable statement of its body finishes.

Local Variables

Variables used for the first time inside of a function definition are called **local** because their use is limited to within the definition of that function. In other words, attempts to access that variable from outside the function definition will fail. The **scope** of a variable is the part of the program in which the variable may be accessed, so the scope of a local variable in Python is the function definition in which it is used. Variables defined outside of all function definitions are called **global**; we will rarely use global variables in this text.

Parameters in a function definition have the same scope as local variables: they may not be accessed outside of the function definition.

In Listing 3.1, `a` and `b` are local variables in the `main()` function, whereas `myhypot()` does not have any local variables.

Type Conversions

The `input()` function always returns user input as a string. In order to process numeric input, we need to **convert** the string to either an integer or float. Python has the following built-in functions to convert the types we have seen so far:

`int(x)`	Convert `x` to an integer, truncating floats towards 0.
`int(x, b)`	Convert `x` given in base `b` to an integer.
`float(x)`	Convert `x` to a floating-point value.
`str(x)`	Convert `x` to a string.

For the input of numeric data, choose between `int()` and `float()` depending on the context. **Truncation** means that any non-integer fraction is dropped, so, for example, `int(3.975)` returns 3, and `int(-3.975)` is -3.

Testing

An important part of writing functions (and programs in general) is **testing** them to make sure that they compute exactly what you intend. Some basic principles of testing are:

Test all important families of inputs. For example, include both positive and negative input values if they are appropriate. Test with both integers and floating point values.

Test known cases. Use input values that give outputs you can predict.

Test boundaries. For example, be sure your function behaves correctly at minimum and maximum input values (when appropriate).

Testing is probably the most important tool we have to improve software quality.

Exercises

3.1 Use Listing 3.1 to answer these questions:

(a) Describe in your own words the subtask that the `myhypot()` function is designed to accomplish.

(b) Which line or lines of code define the body of the `main()` function?

(c) Is there a **`return`** statement inside `main()`? If not, when does `main()` return, and what value does it return?

(d) Which line of code contains a call to the `main()` function? What happens if you try to run the program with that line of code deleted? Explain the result.

3.2 Describe the result if line 6 of Listing 3.1 is replaced with the single line:

```
hyp = sqrt(x ** 2 + y ** 2)
```

Explain the behavior you observe.

3.3 Does the alternate version of the `myhypot()` function on page 15 contain any local variables? If so, identify them; if not, explain why not.

3.4 Use Listing 3.1 to answer these questions:

(a) List important families of inputs to test the `myhypot()` function. Test the program on values from each family and report the results.

(b) List two sets of test values for the `myhypot()` function that fall in the category of known cases, where you know ahead of time what the results should be. Test the program on those values and report the results.

(c) Does `myhypot()` have boundary cases to test? If it does, give them; if not, explain why not.

3.5 Modify Listing 3.1 to call `myhypot(a, b)` before the `print()` statement, storing the result in a new local variable, and then use the local variable in the `print()`. Discuss the tradeoffs.

3.6 Consider Listing 1.1 and its extension to include circumference in Exercise 1.2. Rewrite that version of `circle.py` to ask the user for the radius and use three functions for the area, circumference, and main program.

3.7 Rewrite Exercise 2.9 to ask the user for input and use two functions: one for computing the average and one for the main program.

3.8 Rewrite Exercise 2.10 to ask the user for input and use three functions for the area, perimeter, and main program. How many parameters will the `area()` and `perimeter()` functions need?

3.9 Rewrite Exercise 2.11 to ask the user for input and use three functions for the volume, surface area, and main program.

3.10 Rewrite Exercise 2.12 to ask the user for input and use three functions for the volume, surface area, and main program.

3.11 Write a program `annulus.py` that asks the user for an inner and outer radius, and then calculates and prints the area of an annulus with those dimensions. (Look up annulus if you do not know what one is.) Use a function to calculate the area of an annulus that itself calls a function that computes the area of a circle.

3.12 Write a program `shell.py` that calculates and prints the volume of a spherical shell, given its inner and outer radii. Use a function to calculate the voume of a shell that itself calls a function that computes the volume of a sphere.

3.13 Write a program `temp.py` that asks the user for a temperature in degrees Fahrenheit, and then calculates and prints the corresponding temperature in degrees Celsius. Use a function to perform the conversion.

3.14 Modify the previous exercise to also compute and display the equivalent temperature in degrees Kelvin. Use a separate conversion function (or two).

3.15 Write a program `heart.py` that asks for a person's age in years y and then estimates his or her maximum heart rate in beats per minute using the formula $208 - 0.7y$. Use appropriate functions.

3.16 Write a program `heron.py` that asks the user for the lengths of the sides of a triangle (a, b, and c) and then computes the area of the triangle using Heron's formula. (Look up Heron's formula if you do not remember it.) Use appropriate functions.

3.17 Write a program `interest.py` to calculate the new balance of a savings account if interest is compounded yearly. Ask the user for the principal, interest rate, and number of years, and then display the new balance. Use the formula $P(1+r)^t$ and appropriate functions.

Enter interest rates as decimals: for example, a rate of 4% should be entered as 0.04. Use your program to determine the number of years it takes a principal to double with interest rate 1.5%.

3.18 Write a program `compound.py` to calculate the new balance of a savings account if interest is compounded n times per year. Ask the user for the principal, interest rate, number of years, and number of compounding periods per year, and display the new balance. Use the formula $P(1+\frac{r}{n})^{nt}$ and appropriate functions.

Use your program to determine the difference over 20 years on a \$1000 balance between compounding yearly and compounding monthly at 8% interest.

3.19 Write a program `vp.py` that asks the user for the temperature t in degrees Celsius and then displays the estimated vapor pressure of water vapor in millibars at that temperature using the approximation

$$6.112e^{\frac{17.67t}{t+243.5}}$$

Use appropriate functions. Use your program to estimate the temperature at which the vapor pressure is approximately 10 mb.

Chapter 4

Repetition: For Loops

To this point, our Python programs have been limited to tasks that could be accomplished by a calculator. **Repetition**, the ability to repeat any section of a program, adds a surprising amount of new power.

Listing 4.1: Harmonic Sum

```
# harmonic.py

def harmonic(n):
    # Compute the sum of 1/k for k=1 to n.
    total = 0
    for k in range(1, n + 1):
        total += 1 / k
    return total

def main():
    n = int(input('Enter a positive integer: '))
    print("The sum of 1/k for k = 1 to", n, "is", harmonic(n))

main()
```

This program computes the **harmonic sum**

$$1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \cdots + \frac{1}{n} = \sum_{k=1}^{n} \frac{1}{k}$$

Type it in and run it a few times to get a feeling for what it does and how it works. Try to follow the steps that are executed. This program contains several new features, including the **for** loop, the `range()` function, and the use of `+=`. We will examine each of these components in turn.

For Loops

A **for** loop is one of the two main tools in Python that allow programs to repeat. (The other is **while**, which you will see in Chapter 8.) They are most

useful when you know ahead of time how many times the loop needs to run. The syntax of a **for** loop looks like this:

```
for <variable> in <sequence>:
    <body>
```

The **for loop** works in this way: for each item in the `<sequence>`, the `<variable>` is assigned to have the value of that item and then the loop `<body>` is executed. Thus, the loop will be executed once for each item in the sequence.

A **sequence** in Python is an ordered set of elements. The simplest type of sequence is called a **list**, in which the elements are listed inside square brackets. We will study lists in depth beginning in Chapter 14.

For example, this loop:

```
for i in [8, 3, 0]:
  print(i, i**2)
```

will produce this output:

```
8 64
3 9
0 0
```

The loop body in this case executes three times, once each with `i` having the value 8, 3, and then 0.

Range

The built-in `range()` function is often used inside Python **for** loops instead of writing out long lists. Technically, `range()` returns an **iterable**, which provides the precise thing a **for** loop expects in the place we have referred to as a sequence.

The `range()` function may be called in three different ways. In all three versions, the parameters must be integers.

`range(stop)`	Begin at 0. Take steps of size 1. End just before `stop`.
`range(start, stop)`	Begin at `start`. Take steps of size 1. End just before `stop`.
`range(start, stop, step)`	Begin at `start`. Take steps of size `step`. End just before `stop`.

$\Longrightarrow$ Caution: These are not intuitive at first. Ranges start at 0 unless you specify otherwise, and they end *before* the stopping point you provide.

To see the elements that a `range()` will iterate over, you can use the `list()` type converter:

`list(x)`	Convert `x` to a list.

For example,

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

Notice that with a step size of −1, the sequence still stops *one before* the `stop` we provided.

As you learn more Python, you will find that the `range()` function is written in this way for good reasons.

Assignment Shorthands

We still need to explain the `+=`, although you may have figured it out by now. It is simply a shorthand for a common type of assignment statement:

`x += y`	Equivalent to `x = x + y`.
`x -= y`	Equivalent to `x = x - y`.
`x *= y`	Equivalent to `x = x * y`.
`x /= y`	Equivalent to `x = x / y`.

Accumulation Loops

Listing 4.1 contains an example of an **accumulation loop**, where one of the variables, known as the **accumulator**, gradually accumulates some quantity as the loop runs. In this case, the accumulator is the variable `total`. It begins with the value 0 in line 5, and then each time the loop runs, the code in line 7 adds a bit more to the total.

All accumulation loops look something like this:

```
<accumulator> = <starting value>
loop:
    <accumulator> += <amount to add>
```

After the loop finishes, `<accumulator>` contains the accumulated value. Accumulation loops may also accumulate via multiplication rather than addition. For example, compare Exercise 4.16 with Exercise 4.17.

⟹ Caution: Do not use `sum` as an accumulator variable name, because it is a built-in Python function.

Runaway Loops

Now that we have programs that repeat, we may write a program that runs much too long. If you need to stop a program, use CTRL-C or look for an option to restart your interpreter or shell.

Exercises

4.1 Use Listing 4.1 to answer these questions:

(a) Explain the apparent purpose of comments like the one in line 4.

(b) Explain the use of `n + 1` instead of `n` in line 6.

(c) Give two test values of `n` for which you can easily predict the correct output. Give the outputs for those cases.

(d) Conjecture what happens to the harmonic sum for very large `n`.

4.2 List all local variables used in Listing 4.1 and describe the scope of each.

4.3 Determine the elements that will be iterated over for each of these `range` expressions:

(a) `range(10)`

(b) `range(5, 10)`

(c) `range(10, 5)`

(d) `range(3, 10, 2)`

(e) `range(10, 0, -1)`

(f) `range(10, 0, -2)`

(g) `range(0, 10, -1)`

(h) `range(0, 1, 0.1)`

4.4 Determine a `range` expression to iterate over each of these sequences:

(a) `0, 1, 2, 3`

(b) `3, 2, 1, 0`

(c) `1, 3, 5, 7, 9, 11`

(d) `10, 27, 44, 61, ..., 197`

(e) `1000, 900, 800, ..., 100`

(f) `2, 4, 6, 8, ..., 200`

4.5 Write a program `quote.py` that prints a short quote of your choice exactly one thousand times. You do not need to use any functions other than `main()`.

4.6 Write a program `powers.py` that prints a table of values of n and 2^n for $n = 1, 2, \ldots, 10$. Do these values look familiar? You do not need to use any functions other than `main()`.

4.7 Write a program `table.py` that prints a table of values of n, $\log n$, $n \log n$, n^2, and 2^n for $n = 10, 20, \ldots, 200$. The `log` function is in the `math`

module. You do not need to use any functions other than `main()`. Does the function $n \log n$ grow more like n or more like n^2? How would you describe the growth of 2^n? What about $\log n$?

4.8 Write a program `circletable.py` that prints a table of the areas of circles of radius $r = 1, 2, \ldots, 10$. Use an area function.

4.9 Write a program `temptable.py` that prints a table of Fahrenheit to Celsius conversions for temperatures between $-30°F$ and $100°F$ at 10 degree intervals. Use a conversion function.

4.10 Write a program `disttable.py` that prints a table of mile to kilometer conversions for distances between 100 and 1500 miles at 100 mile intervals. Write a function to do the conversion. One mile is approximately 1.609 km.

4.11 Modify Exercise 3.15 to write a program `hearttable.py` that prints the age and maximum heart rate for ages between 20 and 60 at 2 year intervals.

4.12 Modify Exercise 3.17 to write a program `interesttable.py` that prints the balance in an account earning simple interest at the end of each year for a given number of years. Ask the user for the starting principal, interest rate, and number of years.

4.13 Rewrite the function from Exercise 3.17 that computes interest compounded yearly to use an accumulator rather than the direct formula.

4.14 Modify Exercise 3.19 to write a program `vptable.py` that prints the estimated vapor pressures for temperatures between -20 and 50 degrees Celsius at 5 degree intervals.

4.15 Write a program `basel.py` that modifies Listing 4.1 to print a table of values of $\displaystyle\sum_{k=1}^{n} \frac{1}{k^2}$ for $n = 10, 100, 1000, \ldots, 10^7$. Use your program to conjecture what happens to this sum as n becomes large. Hints: Use a `range()` to create the exponents, and in addition to the sum, also print the square root of six times the sum.

4.16 Define a function `triangular(n)` that returns the sum $1+2+3+\cdots+n$. Then write a program `triangular.py` that prints a table of values of `triangular(n)` for $n = 1, 2, 3, \ldots, 20$.

4.17 Write a function `myfactorial(n)` that returns the product $1 \cdot 2 \cdot 3 \cdot \cdots \cdot n$. Use an accumulator; do not use the `factorial()` function from the `math` module. Then write a program `factorial.py` that prints a table of values of `myfactorial(n)` for $n = 1, 2, 3, \ldots, 20$. Finally, modify your program to compare the growth of factorials with powers (like n^2) and exponentials (like 2^n). Discuss your results.

4.18 Write a function `pyramid(n)` that returns $1^2 + 2^2 + 3^2 + \cdots + n^2$. Then write a program `pyramid.py` that prints a table of values of `pyramid(n)` for $n = 1, 2, \ldots, 20$. Finally, modify your program to estimate the growth rate of the `pyramid()` function. Does it appear similar to a power function or an exponential? Discuss your results.

Chapter 5

Computer Memory: Integers

It is time to go back to the machine level and look a little more deeply at memory. At a high level, computer memory may be categorized according to concepts such as access speed and size. Definite patterns emerge:

Type	**Access Speed**	**Proximity to CPU**	**Size**	**Volatile**
Register	Fastest	Inside	10's	Y
Caches	Very fast	Adjacent	100's to MB	Y
RAM	Fast	Near	GB	Y
Hard disks	Slow	Far	TB	N

At a lower level, the question is, how is data actually stored in computer memory? In this chapter, we begin to develop an answer by looking at how integers are stored.

Bits and Bytes

Computer memory of all types may be thought of at different levels of interpretation. The bottom level is electronics and physics, which is taught in those courses. We will move up one level of interpretation and begin by thinking of memory as a sequence of electronic on/off switches. Each on/off switch is called a **bit**. A group of 8 bits is called a **byte**.

In abbreviations, a small "b" refers to bits, while capital "B" refers to bytes. So, for example, Mbps refers to megabits per second, while GB refers to gigabytes.

Binary Numbers

We then interpret bits as numbers by thinking of "off" as 0 and "on" as 1. For example,

off	on	on	off	on	off	off	off
0	1	1	0	1	0	0	0

We interpret this number as a number in base two instead of base ten. Now, the numbers that you use every day are **decimal** or **base ten**. Think about

how they work:

$$\begin{array}{cccc} \boxed{2} & \boxed{1} & \boxed{7} & \boxed{4} \\ 1000\text{'s} & 100\text{'s} & 10\text{'s} & 1\text{'s} \\ 10^3 & 10^2 & 10^1 & 10^0 \end{array} = 2 * 1000 + 1 * 100 + 7 * 10 + 4 * 1$$

Binary numbers are **base two** and work the same way except that instead of powers of ten, they use powers of two. Note that with base ten, we use the digits 0–9 (less than ten); with base two, we only use the digits 0 and 1 (less than two). For example,

$$\begin{array}{cccc} \boxed{1} & \boxed{0} & \boxed{1} & \boxed{1} \\ 8\text{'s} & 4\text{'s} & 2\text{'s} & 1\text{'s} \\ 2^3 & 2^2 & 2^1 & 2^0 \end{array} = 1 * 8 + 0 * 4 + 1 * 2 + 1 * 1 = 11$$

Hexadecimal Numbers

Hexadecimal numbers are **base sixteen** and work the same way with powers of sixteen and digits 0–9, A=10, B=11, C=12, D=13, E=14, and F=15. Every four bits can be thought of as a single hexadecimal "digit," since four bits can hold values between 0 and 15. Thus, the byte 01101100 can be represented as `6C` hexadecimal, since 0110 equals 6 and 1100 equals 12, which is `C`.

Python has built-in conversion functions that you may find useful:

`bin(n)`	Binary value of integer `n` (as a string).
`hex(n)`	Hex value of integer `n` (as a string).

Binary strings begin with `"0b"` in Python, while hex strings begin with `"0x"`.

Storing Integers

When a computer is described as "32-bit" or "64-bit," that tells you the basic memory size used by its CPU. Generally, this also gives the size of memory that is used to store an integer. **Unsigned integers** are always greater than or equal to zero and are stored in binary, as described above, using the number of bits given by the architecture. For example, on a 32-bit machine, the unsigned integer 2012 is stored as `0x000007dc`.

Signed integers, on the other hand, may be either positive or negative and so require a more complicated representation. Most computer systems use **two's complement**, which is taught in computer architecture courses.

Memory Sizes

Memory sizes are generally given in terms of bytes, except that so many bytes are involved that usually a prefix is used to indicate the scale at which we are working. Common prefixes are borrowed from the metric system, such as

kilo-, mega-, giga-, and tera-. Unfortunately, these prefixes do not quite mean what they do in the metric system, where they are based on powers of 10:

Prefix	**Value**
kilo-	$10^3 = 1000 = 1$ thousand
mega-	$10^6 = 1{,}000{,}000 = 1$ million
giga-	$10^9 = 1{,}000{,}000{,}000 = 1$ billion
tera-	$10^{12} = 1{,}000{,}000{,}000{,}000 = 1$ trillion

Because computer memory is based on bits, powers of 2 are often used for these sizes instead:

Prefix	**Value**
kilo-	$2^{10} = 1024$
mega-	$2^{20} = 1{,}048{,}576$
giga-	$2^{30} = 1{,}073{,}741{,}824$
tera-	$2^{40} = 1{,}099{,}511{,}627{,}776$

These are close to their metric equivalents but are not exactly the same. This is one reason computer memory is sold in quantities that sometimes look strange.

Notice that $2^{10} = 1024 \approx 1000$. This is a useful fact to help you remember the size of powers of 2.

Exercises

5.1 Using either your own computer or one in a lab, determine the number of registers in its CPU, as well as the size of its caches, RAM, and hard drive (or other long-term storage).

5.2 Give the largest binary value that can be stored in one unsigned byte, along with its decimal and hexadecimal equivalents.

5.3 Give the largest binary value that can be stored in two unsigned bytes, along with its decimal and hexadecimal equivalents.

5.4 Determine the largest unsigned integer that can be stored in a 32-bit machine.

5.5 Determine the largest unsigned integer that can be stored in a 64-bit machine.

5.6 Give the values of these Python expressions:

(a) `hex(25)`
(b) `bin(35)`
(c) `int("0x1C", 16)`
(d) `int("0b10101", 2)`
(e) `int(bin(1000), 2)`
(f) `int(hex(1000), 16)`

5.7 Show how each of these is stored as an unsigned integer in a byte. Write both binary and hexadecimal forms.

(a) 87
(b) 195
(c) 18
(d) 119
(e) 93
(f) 234

5.8 Convert these unsigned binary integers to decimal and hexadecimal.

(a) `0b10010011`
(b) `0b00101101`
(c) `0b01001011`
(d) `0b10011110`
(e) `0b01011100`
(f) `0b11000001`

5.9 Convert these unsigned hexadecimal integers to binary and decimal.

(a) `0x7D`
(b) `0xA1`
(c) `0x59`
(d) `0xBC`
(e) `0x96`
(f) `0x04`

5.10 Write a short program `binhex.py` that prints a table of binary and hexadecimal values for the (decimal) integers 1 through 100. You do not need any functions other than `main()`.

Chapter 6

Selection: If Statements

At both high levels and at the machine level, programs execute statements one after the other. **Selection** statements allow a program to execute different code depending on what happens as the program runs. This flexibility is another key to the power of computation.

Listing 6.1: Centipede

```
# centipede.py

from turtle import *

def centipede(length, step, life):
    penup()
    theta = 0
    dtheta = 1
    for i in range(life):
        forward(step)
        left(theta)
        theta += dtheta
        stamp()
        if i > length:
            clearstamps(1)
        if theta > 10 or theta < -10:
            dtheta = -dtheta
        if ycor() > 350:
            left(30)

def main():
    setworldcoordinates(-400, -400, 400, 400)
    centipede(14, 10, 200)
    exitonclick()

main()
```

This program may be harder to read at first than previous examples. It uses what are called "turtle graphics," explained later in this chapter. Run it, and

then you will begin to see what the code is doing. Use "Restart Shell" from the Shell menu in IDLE if your window ever becomes unresponsive.

The new features in this program include the **if** statement, boolean expressions, and the turtle graphics library. In order to describe **if** statements, we need to begin with boolean expressions.

Boolean Expressions

Python has an additional data type known as **boolean**. Variables that hold this type have only two possible values: `True` or `False`. Generally, you will use boolean values in expressions rather than variables. Thus, a **boolean expression** is an expression that evaluates to either `True` or `False`. Python has the following **comparison operations** that return boolean values:

`x == y`	Equal.
`x != y`	Not equal.
`x < y`	Less than.
`x > y`	Greater than.
`x <= y`	Less than or equal.
`x >= y`	Greater than or equal.

$\Longrightarrow$ Caution: To test for equality in Python, you must use `==` rather than `=`.

$\Longrightarrow$ Caution: Avoid using `==` or `!=` to compare floats; instead, try to use an inequality if you can. The reason is that floating-point calculations are not always exact. See Chapter 9 for an explanation.

Boolean expressions may be combined with these **boolean operations**:

`P` **`and`** `Q`	`True` if both `P` and `Q` are `True`; otherwise, `False`.
`P` **`or`** `Q`	`True` if either `P` or `Q` (or both) are `True`; otherwise, `False`.
`not` `P`	`True` if `P` is `False`; otherwise, `False`.

The **and** and **or** operations **short-circuit** their evaluation, meaning, for example, that when evaluating `P` **`and`** `Q`, if `P` is `False`, then there is no need to evaluate `Q` because the result must be `False`.

If Statements

Almost all programming languages (including machine languages) have some form of **if** statement. In Python, the statement has essentially three forms. The simplest is:

```
if <boolean>:
    <body>
```

In this form, the `<boolean>` expression is evaluated, and if it is `True`, then `<body>` is executed. If the expression is `False`, then `<body>` is not executed.

An **if** statement may contain an optional **else** clause, which contains alternative code to run when the boolean expression is `False`:

```
if <boolean>:
    <body1>
else:
    <body2>
```

In this case, `<body1>` is executed if the boolean expression is `True`; otherwise, `<body2>` is executed.

Finally, a sequence of tests may be checked by using the **elif** option:

```
if <boolean1>:
    <body1>
elif <boolean2>:
    <body2>
elif <boolean3>:
    <body3>
...
else:
    <bodyN>
```

Here, if `<boolean1>` is `True`, then `<body1>` is executed; otherwise, `<boolean2>` is evaluated, and if it is `True`, `<body2>` is executed; and so on. Later tests are checked only if all preceding tests are `False`.

Python Turtle Module

Turtle graphics is a type of computer graphics that draws relative to the position of a "turtle" on the screen. The turtle holds a pen, and if the pen is down when the turtle moves, then a line will be drawn. The turtle may also move with the pen up or initiate other types of drawing such as "stamping" its own shape or drawing dots.

Listing 6.1 uses the following functions from the Python `turtle` module:

`penup()`	Stop drawing turtle path.
`forward(x)`	Move forward `x`.
`left(theta)`	Turn left angle `theta` (default is degrees).
`stamp()`	Draw turtle shape at current location.
`clearstamps(n)`	Delete first `n` stamps (if `n` is positive).
`ycor()`	y coordinate of current location.
`setworldcoordinates(llx, lly, urx, ury)`	Set lower-left and upper-right coordinates.
`exitonclick()`	Close turtle window when clicked.

Using the Python Documentation

You can imagine that the `turtle` module must provide many other functions in addition to those listed above. You may also have questions about exactly how those functions work. The **Python documentation** is online, extensive, and provides information like this and much more.

From the "Documentation" link at `http://www.python.org`, two links will be particularly useful: the Tutorial and the Library Reference.

Tutorial provides informal descriptions of how most things work in Python. Use it when you start to learn a new topic.

Library Reference is a good place to look up specific reference information. For example, at the time of this writing, the complete list of functions in the `turtle` module is in Section 23.1 of the Library Reference for Python 3.2.

Be sure to use the documentation set that matches your version of Python.

Import Star

Python provides a star form of the **`import`** statement that is helpful when you need to use many names from the same module:

```
from turtle import *
```

This imports all names from the module `turtle`. Use the star sparingly; otherwise, you may find that a module has imported names that you didn't anticipate.

Multiple Return Values

Occasionally, it is useful to have a function return more than one value:

```
return <expression1>, <expression2>, ...
```

The mechanism used to accomplish this will be described later in Chapters 16 and 19.

Exercises

6.1 Explain the difference between = and == in Python.

6.2 Evaluate these boolean expressions:

(a) `1.4 ** 2 > 2`

(b) `6 // 7 == 1 // 7`

(c) `10 >= 11 or 11 < 12`

(d) `3 > 1 and 4.25 > 9 / 2`

6.3 Write the mathematical condition "x is in the interval $(2, 3]$" as a Python boolean expression.

6.4 Describe the circumstances when the **or** operation can short-circuit, and briefly explain why that behavior is correct.

6.5 List all local variables used in Listing 6.1 and describe the scope of each.

6.6 Determine the range of x and y values for the turtle screen in Listing 6.1. Use `print(xcor(), ycor())` to verify your answer.

6.7 Determine the initial location and orientation of the turtle in a turtle graphics program.

6.8 Describe the effect of changing each of these quantities in Listing 6.1:

(a) `length`

(b) `step`

(c) `life`

(d) `dtheta`

(e) The 350 in line 18

(f) The 10 and -10 in line 16

6.9 Is it possible to turn right using the `turtle left()` function? Explain why or why not.

6.10 Use Listing 6.1 to answer these questions:

(a) Describe the effect of removing the **if** statement (and its body) at line 14. Explain the result.

Hint: an easy way to do this is to put comment symbols at the beginning of those two lines, thereby changing the code into comments. This is called **commenting out** a section of code.

(b) Describe the effect of removing the **if** statement (and its body) at line 16. Explain the result.

(c) Describe the effect of removing the **if** statement (and its body) at line 18. Explain the result.

6.11 Modify Listing 6.1 to keep the pen down rather than up. Describe the effect.

6.12 Modify line 16 of Listing 6.1 to use the built-in absolute value function `abs()` instead of an **`or`** expression.

6.13 Modify Listing 6.1 to prevent it from escaping across all four sides.

6.14 Modify Listing 6.1 to produce an interesting different behavior of your choice.

6.15 Write an interesting turtle graphics program of your choice.

6.16 Write a function `grade(score)` that returns the corresponding letter grade for a given numerical score. Use 90 or above for an A, 80 for a B, etc. Write a `main()` that tests your function.

6.17 Write a function `mymax2(x, y)` that returns the larger of `x` and `y`. Do not use the built-in Python function `max()`. Write a `main()` that tests your function.

6.18 Write a function `mymax3(x, y, z)` that returns the largest of `x`, `y`, and `z`. Do not use the built-in Python function `max()`. Write a `main()` that tests your function.

6.19 Write a function `median3(x, y, z)` that returns the middle value among `x`, `y`, and `z`. (If two of the values happen to be the same, that value is the median.) Do not use any built-in Python sorting functions. Write a `main()` that tests your function.

6.20 Write a function `sort3(x, y, z)` that returns the three values `x`, `y`, and `z` in **sorted order** `(a, b, c)`, where $a \leq b \leq c$. Do not use any built-in Python sorting functions, but you may put **`if`** statements inside of other **`if`** statements. Write a `main()` that thoroughly tests your function.

6.21 Rewrite the function `median3(x, y, z)` of Exercise 6.19 using the function `sort3()` from Exercise 6.20. Use multiple assignment (see page 95) to store the return values of `sort3()`.

6.22 Write a function `myabs(x)` that returns $|\,x\,|$, the absolute value of `x`, which is given by:

$$|\,x\,| = \begin{cases} x & \text{if } x \geq 0 \\ -x & \text{otherwise} \end{cases}$$

Do not use the built-in Python function `abs(x)`. Write a `main()` that tests your function.

6.23 Rewrite the `myfactorial(n)` function from Exercise 4.17 to use this recursive definition:

$$n! = \begin{cases} 1 & \text{if } n = 0 \text{ or } 1 \\ n * (n-1)! & \text{otherwise} \end{cases}$$

You may assume `n` is not negative. Write a `main()` that tests your function.

6.24 Write a function `solvequadratic(a, b, c)` that returns the solution(s) of the quadratic equation $ax^2 + bx + c = 0$. Use the discriminant $d = b^2 - 4ac$ to determine whether there are 0, 1, or 2 real roots. Python has a built-in `complex()` function if you want to return complex roots.

6.25 Write a function `zone(age, rate)` that returns a description of a person's training zone based on his or her age and training heart rate, `rate`. The zone is determined by comparing `rate` with the person's maximum heart rate `m`:

	Training Zone
`rate` $\geq .90$ `m`	interval training
$.70$ `m` $\leq$ `rate` $< .90$ `m`	threshold training
$.50$ `m` $\leq$ `rate` $< .70$ `m`	aerobic training
`rate` $< .50$ `m`	couch potato

Use the function from Exercise 3.15 to determine `m`. Write a `main()` to ask the user for input and display the result.

This page intentionally left blank

Chapter 7

Algorithm Design and Debugging

Developing Algorithms

As the problems we tackle become more complex, it becomes harder to quickly write out programs to solve them, and we need to devote correspondingly more time to planning and designing solutions as opposed to writing code. At this stage, the difficulty is generally in designing an algorithm capable of solving the problem.

An **algorithm** is just a specific sequence of steps that will solve some problem. Prior to writing code, programmers often use **pseudocode**, which is a hybrid of English (or other language) and programming code. Thinking in pseudocode allows you to focus on how to solve the problem rather than language details or syntax. However, to be useful, the pseudocode must be specific enough that it can be translated into a working program.

Recipes are good everyday examples of algorithms. They can be written at different levels of detail, depending on the expertise or experience of the cook. What can be difficult about programming is that the computer almost always requires a *more precise* level of detail than we are used to providing.

The following guidelines may be helpful as you begin to design algorithms:

Know what the language can do. To some extent, you have to limit your thinking to what a program can do; otherwise, you may write pseudocode that cannot be translated into a program.

Recognize patterns. For example, accumulation loops occur in many different contexts (as you will see). Patterns like it expand the vocabulary you can think in.

Think top-down or bottom-up. Top-down design starts with large process steps and gradually breaks each one down until there is enough detail to implement it. Bottom-up works in reverse, beginning with relatively detailed tasks, and then putting those together until there is a complete solution. Combining top-down with bottom-up thinking can be quite powerful.

Comparing Algorithms

Once we have an algorithm, how do we know if it is a good one? The main criterion, whether we are beginners or professionals, is the same:

Does it work?

This simple question hides enormous complexity, but keep it firmly in mind.

Beyond the question of producing correct results, algorithms and programs are also judged on:

Efficiency. Does the program run in a reasonable amount of time? Could it be faster?

Space. Does the program use a reasonable amount of memory, or is it unnecessarily taxing system resources?

Elegance. Does the program represent an elegant solution to the problem, or does it feel like a maze with no exit?

Courses in data structures and algorithm analysis explore these questions in more depth.

Debugging

As programs become more complicated, it also becomes more difficult to analyze their behavior, particularly when problems arise. **Debugging** is the art of finding and removing errors or **bugs** in programs.

Arts often sound like mysteries, especially when you are a beginner in the field, but there is also a science to debugging. The science involves becoming *systematic* rather than random when trying to understand program behavior.

$\Longrightarrow$ Caution: Random changes to programs produce random results.

The key to systematic debugging is to take advantage of the fact that programming is one of the few disciplines that offers quick and accurate feedback to your problem-solving efforts. Imagine a math problem that told you whether you had solved it or not—essentially, that is what a computer does every time you run a program.

Here is one way[1] to systematically debug a program:

Experience what the program does when it runs. In other words, *pay attention* to the program's behavior. This step sounds obvious, but it is easy to overlook and is a necessary precondition to making intentional progress. If your program does not run, the Python interpreter should help you find syntax errors. Fix those first.

[1] This outline is based on James Zull's *The Art of Changing the Brain* [6], which gives these steps as an outline for learning. The power of this particular model is that the outline is based on brain structure and how the brain seems to process information.

Reflect on the behavior you observe. Take a minute to make sure you understand what you have seen. At the end of this step, you should be able to give a precise description of the (incorrect) behavior of your program in your own words.

Hypothesize what might be causing the program's current behavior. This is the key step. You need to discover *why* the program is doing what it is doing. That will often tell you how to fix it.

Test your hypothesis. Make an intentional change (not a random one), and see if the program responds in the way you expect. One of the easiest ways to test an idea is to insert extra `print()` statements that let you see the values of important variables.

Repeat until the program works.

Practice these steps regularly, and you will develop effective debugging habits.

Exercises

7.1 Write an algorithm to make a favorite food for:

(a) A younger sibling who knows the food but is new to cooking

(b) A college student who does not know the food but cooks regularly

7.2 Write an algorithm to get to some location in your home town for:

(a) A local friend

(b) A friend from out of town

7.3 You have a list of numbers which provide access to one element at a time. Write an algorithm in pseudocode to:

(a) Find the smallest number in the list.

(b) Find the largest number in the list.

(c) Sort the list in increasing order.

7.4 Research the bug associated with Grace Hopper. Summarize and discuss what you find.

This page intentionally left blank

Chapter 8

Repetition: While Loops

While loops allow programs to repeat without necessarily knowing ahead of time how many times the loop will run. Like selection, this allows programs to adapt and run more or less code, depending on the circumstances.

Consider the task of printing a table of prime numbers. An integer n is **prime** if it is greater than 1 and its only positive divisors are 1 and itself. This definition leads to the following program:

Listing 8.1: Prime Numbers

```
1  # primes.py
2
3  def isprime(n):
4      # Return True if n is prime.
5      return n > 1 and smallestdivisor(n) == n
6
7  def smallestdivisor(n):
8      # Find smallest divisor (other than 1) of integer n > 1.
9      k = 2
10     while k < n and not divides(k, n):
11         k += 1
12     return k
13
14 def divides(k, n):
15     # Return True if k divides n.
16     return n % k == 0
17
18 def main():
19     for n in range(2, 100):
20         if isprime(n):
21             print(n, end=" ")
22     print()
23
24 main()
```

The definition of the `isprime()` function parallels the definition of prime number in a nice way and reads like pseudocode. In addition to the **`while`** loop, this program uses the modulus operator `%` and the `end=` option of `print()` for the first time.

While Loops

The syntax of a **`while`** loop is almost identical to that of an **`if`** statement.

```
while <boolean>:
    <body>
```

The `<boolean>` expression is evaluated, and if it is `True`, then `<body>` is executed. (So far, this is identical to an **`if`** statement.) After the body executes, `<boolean>` is evaluated again, and if it is still `True`, then the body executes again. This is repeated until the boolean expression is `False`.

Unlike a **`for`** loop, it is distinctly possible for a **`while`** loop to be **infinite**. Use CTRL-C or look for an option to restart your interpreter or shell if you execute an infinite loop.

Mod Operation

The **modulo** or **mod** operation finds the remainder when one integer is divided by another.

`n % m`	Remainder when `n` is divided by `m`.

So, for example, `k` divides `n` evenly if the remainder is 0; i.e., if `n % k == 0`.

Print Options

The Python `print()` function has two options that give you somewhat more control over output:

`print(..., end=s)`	Print with string `s` at end instead of newline.
`print(..., sep=s)`	Print with string `s` between expressions.

Both options may be set at the same time.

Tracing Code by Hand

You may have found it hard to follow exactly what is happening in Listing 8.1. One reason might be its use of several functions, another could be the **`while`** loop in `smallestdivisor()`. Tracing code by hand can help overcome both of these difficulties. When we **trace** code by hand, we try to simulate on paper what the interpreter is asking the CPU to do.

For example, we might trace the evaluation of `isprime(9)` like this:

```
isprime(9)
    → smallestdivisor(9)
        → divides(2, 9)   returns F
        → divides(3, 9)   returns T
    returns 3
returns F
```

To follow the operation of `smallestdivisor()`, it may also be helpful to track the values of its variables, like this for the call `smallestdivisor(25)`:

n	k	
25	2	
	3	
	4	
	5	Returns 5

There is nothing special about these ways of writing the traces; the point is just to find a helpful way of simulating the program's execution.

Nested Loops

Although it is not immediately obvious, Listing 8.1 runs its **`while`** loop *inside* of another loop—the **`for`** loop in `main()`. That means the entire **`while`** loop from `smallestdivisor()` runs once every time the body of the **`for`** loop runs.

Any time one loop is run inside of another loop, the loops are called **nested**. As you may imagine, nested loops can have a significant impact on program performance.

Exercises

8.1 Calculate these values by hand:

(a) `23 % 4`
(b) `15 % 7`
(c) `52 % 19`
(d) `219 % 105`
(e) `1738 % 3`
(f) `418 % 2`
(g) `5033 % 2`
(h) `128 % 10`
(i) `741 % 5`

8.2 Trace the execution of each of these function calls:

(a) `isprime(7)`
(b) `isprime(27)`
(c) `isprime(1024)`
(d) `smallestdivisor(100)`
(e) `smallestdivisor(81)`
(f) `smallestdivisor(49)`

8.3 Write a function `is_even(x)` that returns the boolean value `True` if `x` is even; otherwise, it returns `False`.

8.4 Write a function `is_odd(x)` that returns the boolean value `True` if `x` is odd; otherwise, it returns `False`.

8.5 Determine the output of these two fragments of Python code. Which of the two has nested loops? How many `print()` statements are executed in each case?

(a)
```
for i in range(4):
    print(i)
for j in range(3):
    print(j)
```

(b)
```
for i in range(4):
    print(i)
    for j in range(3):
        print(j)
```

8.6 Write an infinite loop.

8.7 Use Listing 8.1 to answer these questions:

(a) Explain the purpose and effect of the `end=" "` in line 21. What happens if it is omitted?
(b) Explain the purpose and effect of the `print()` statement in line 22. What happens if it is omitted?
(c) Determine the value of the function call `smallestdivisor(1)`.
(d) Is there any way for the loop in `smallestdivisor()` to be infinite? Explain your answer.
(e) Discuss the tradeoffs in having `divides()` as a separate function.

8.8 Consider this variation of the `divides()` function from Listing 8.1:

```
def divides(k, n):
    if n % k == 0:
        return True
    else:
        return False
```

Discuss the tradeoffs between writing the function in this way compared with the original version.

8.9 Modify the `smallestdivisor()` function in Listing 8.1 to use a **`for`** loop instead of a **`while`** loop. Discuss the tradeoffs. Hint: a function may **`return`** at any time.

8.10 If an integer n is not prime, then at least one of its divisors must be less than or equal to $\sqrt{n}$. (Think about why.) Use this fact to improve the performance of the `smallestdivisor()` function in Listing 8.1.

8.11 Write a program `change.py` that asks the user for a number of cents and computes the number of dollars, quarters, dimes, nickels, and pennies needed to make that amount, using the largest denomination whenever possible. For example, 247 cents is 2 dollars, 1 quarter, 2 dimes, and 2 pennies. Use `//` when you intend to use integer division. You do not need to write any functions other than `main()`.

8.12 Write a program `guesser.py` that guesses an integer chosen by the user between 1 and 100. After each guess, the user indicates if the guess was too high, too low, or correct. Your program should use as few guesses as possible. You do not need to write any functions other than `main()`.

8.13 Modify `guesser.py` from Exercise 8.12 to add these features:

(a) Allow the user to play more than once.

(b) Report the total number of guesses taken after each round.

(c) Allow the user to specify the upper limit, instead of always using 100.

(d) Have your program sound confident by predicting the maximum number of guesses it will take. The `log()` and `ceil()` functions from the `math` module may be helpful.

8.14 Write a program `savings.py` that calculates the number of years it will take to reach a savings goal given a starting principal and interest rate. Write appropriate functions, ask the user for input, and display the result. Hint: use an accumulator rather than a formula.

8.15 Write the function `intlog2(n)` that returns the largest integer k such that $2^k \leq n$. For example, `intlog2(20) = 4` because $2^4 = 16$ is the largest power of 2 less than or equal to 20, and `intlog2(32) = 5` because $2^5 = 32$ is the largest power of 2 less than or equal to 32. Write a `main()` to test your function. Do not use any library functions inside your `intlog2()`.

8.16 Modify Listing 4.1 to ask the user for a number `m` and then compute the smallest `n` for which the harmonic sum $\displaystyle\sum_{k=1}^{n} \frac{1}{k}$ is greater or equal to `m`. Be careful to test your program only with small `m`.

8.17 Write a function `gcd(m, n)` that calculates the **GCD** (greatest common divisor) of `m` and `n`, which is the largest positive `k` that divides both `m`

and `n`. Use **Euclid's algorithm** to calculate the gcd. Here is one (very succinct!) way to describe it:

```
Replace m with n, and n with m % n until n is 0.
```

Once `n` becomes 0, the gcd is in `m`. Write a program `gcd.py` with a `main()` that uses nested loops to test your function thoroughly.

Project: Newton's Method

Newton's method is a powerful technique for numerically computing the zeros of differentiable functions. It is an **iterative technique**, meaning that instead of applying a formula to directly compute the answer, it repeatedly tries to compute a better approximation until the desired accuracy is reached. Iteration is just a fancy word for repetition, which we know how to do in Python using **`for`** loops and **`while`** loops.

Newton's method gives a surprisingly simple algorithm for computing $\sqrt{k}$.

Listing 8.2: Newton's Method (Algorithm)

```
# To compute sqrt(k)
Begin with an initial guess x.
Repeat until desired accuracy is reached:
    Replace x with the average of x and k/x.
```

That's all there is to it. At the end of the loop, `x` will be approximately equal to $\sqrt{k}$.

This is an algorithm in pseudocode, but it isn't yet an executable program. Your task will be to convert this algorithm into a working Python program.

Here are some new bits of Python that may be helpful:

Scientific notation Floats may be specified in Python with an "e" before the exponent. For example, `2.914e6` represents $2.914 * 10^6 = 2914000.0$.

Absolute value Python provides the built-in function:

`abs(x)`	Absolute value of `x`.

The reason absolute value is useful is that it gives an easy measure of how close together two numbers are: the distance between x and y is $|\, x - y \,|$.

One last piece of advice that addresses a question you may have already thought of. How is it possible to know how close x is to $\sqrt{k}$ without knowing the value of $\sqrt{k}$ ahead of time? In other words, how do we know when we are close enough? The answer is to compare x^2 with k instead of x to $\sqrt{k}$. This will not tell you how close x is to the actual root, but it gives you some measurement of accuracy.

Exercises

1. Write a function `mysqrt(k)` that uses Newton's method to approximate $\sqrt{k}$ using an accuracy of 10^{-10}. Do not use the library `sqrt()` function. Write a program `newton.py` that uses your function to display the square roots of 2, 3, 4, ..., 20.

2. Compare the results of your `mysqrt(k)` function with the `sqrt()` function in the `math` module.

3. Experiment with different initial guesses in your program and report the results.

4. Experiment with different values for the required accuracy and report the results.

Chapter 9

Computer Memory: Floats

Predict what this code will do when it runs:

```
x = 0
while x != 1:
    print(x)
    x += 0.1
```

Then run it. Are you surprised? Ctrl-C, in case you have forgotten.

What is going on here? Isn't 0.1 the same as 1/10?

The explanation is subtle, but it affects every floating-point computer program in the world, including some used in life-or-death situations. The basic issue is that, like integers, floating-point numbers are stored in binary rather than decimal form.

Binary Fractions

In base ten, you are familiar with the fact that 1/3 has an infinitely-repeating decimal form, 0.33333333 . . .. If we used base ten computers, you would probably be nervous about storing the fraction 1/3 as a float, because computers are *finite* and so wouldn't be able to store all of its digits—they would be cut off at some point.

Well, computers use base two, and 1/10 has an infinitely repeating binary form:

$$\frac{1}{10} = 0.0001100110011\ldots$$

This is because fractional values are converted to binary in the same way as integers, except that they use negative powers of 2. For example,

$$\begin{array}{ccccccccc} 0 & . & \boxed{1} & \boxed{0} & \boxed{1} & \boxed{1} \\ & & 2^{-1} & 2^{-2} & 2^{-3} & 2^{-4} \end{array} = 1 * \frac{1}{2} + 0 * \frac{1}{4} + 1 * \frac{1}{8} + 1 * \frac{1}{16} = \frac{11}{16}$$

Thus, 1/10 cannot be stored exactly as a floating-point value in a binary computer; it will be cut off at some point and therefore be a little bit off.

The exact details of how floating-point values are stored in memory are beyond the scope of this course, but you will find them in courses on computer architecture and numerical analysis.

Moral: Use Inequalities with Floats

Because so many floating-point values cannot be stored exactly in memory, testing floats using either `==` or `!=` is risky. Use inequalities (`<`, `<=`, `>`, `>=`) whenever possible.

Exercises

9.1 Convert the following decimal values to binary. Indicate which can or cannot be stored precisely as floats.

(a) 0.25
(b) 0.375
(c) 5.125
(d) 10.4375
(e) 0.3
(f) 0.5

9.2 Convert the following binary values to decimal.

(a) 0.1
(b) 0.0101
(c) 0.111
(d) 10100.01
(e) 111.1011
(f) 1000.10001

9.3 Determine whether or not the fraction 1/3 can be stored exactly as a binary floating-point value.

9.4 Fix the code at the beginning of this chapter so that it terminates as apparently intended.

9.5 Research the connection between the storage of floats and the performance of Patriot missiles in the 1991 Gulf War.

9.6 Discuss the advantages and disadvantages of using floating-point variables to store monetary values.

Chapter 10

Simulation

Consider the problem of finding the area under the curve $f(x) = e^{-x^2}$ between $x = 0$ and $x = 2$:

Figure 10.1: Graph of $f(x) = e^{-x^2}$.

Even if you know calculus, this is a difficult task. However, a relatively simple idea will allow us to approximate the area.

The idea begins with putting a box around the graph that contains it completely. For a function that stays positive, this means finding a maximum value m; in this case, we can use $m = 1$:

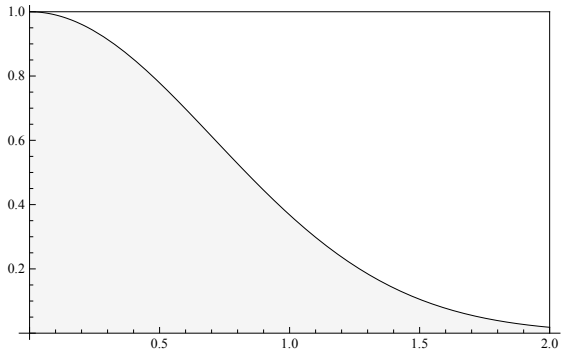


Figure 10.2: Box containing area.

Now imagine throwing random darts into the box. Some will land below the graph, in the area we want to measure, and some will miss and be above the

graph. If we throw enough darts and measure the fraction of those that hit the area we want, then we can approximate the area under the curve:

Hits 88
Misses 112

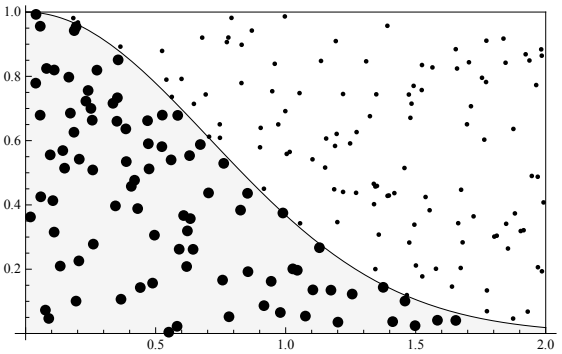


Figure 10.3: Darts thrown in box.

Since 88 of the 200 darts hit under the curve, and the total area of the box is 2, the area under the curve is approximately

$$\frac{88}{200} * 2 = 0.44 * 2 = 0.88$$

This technique is called Monte Carlo integration, and is an example of a more general class of techniques known as **Monte Carlo simulations**. Monte Carlo integration is used widely in computer graphics because the integrals that describe physically accurate lighting cannot be evaluated analytically.

Listing 10.1: Monte Carlo Integration

```
# montecarlo.py

from random import uniform
from math import exp

def estimate_area(f, a, b, m, n=1000):
    # Estimate area under f over [a, b] for f positive and <= m.
    hits = 0
    total = m * (b - a)
    for i in range(n):
        x = uniform(a, b)
        y = uniform(0, m)
        if y <= f(x):
            hits += 1
    frac = hits / n
    return frac * total

```

```
18 def f(x):
19     return exp(-x ** 2)
20
21 def main():
22     print(estimate_area(f, 0, 2, 1))
23
24 main()
```

Most of this code should be familiar. The new features are the function calls to produce random values, the use of a default argument, and passing a function as a parameter.

Random Numbers

The main new feature we need in order to write simulations is a source of randomness. Most computer games also use some form of randomness. The `random` module in Python provides several functions that return pseudorandom values:

`random()`	Random float from $[0, 1)$.
`uniform(a, b)`	Random float from $[a, b]$.
`randint(a, b)`	Random integer from $[a, b]$.
`randrange(start, stop, step)`	Random integer from `range(start, stop, step)`.

Think of these as working in such a way that any of their possible return values are equally likely.

Default Arguments

Python allows a function to specify **default arguments** in its definition, as in line 6 of Listing 10.1. The default value is put after an equals sign that follows the parameter name, and is used if the argument is not specified at the time of the function call.

Functions as Parameters

Listing 10.1 illustrates the fact that Python allows functions as parameters to other functions. The name of the function to use is passed as an argument when the function is called. In line 22, we pass the function named `f` to `estimate_area`. That function is then used in line 13, where it is called with whatever the current value of `x` is. This is quite a powerful feature.

Exercises

10.1 Use Listing 10.1 to:

(a) Describe what the parameter `n` controls in the `estimate_area()` function.

(b) Describe the role of the variable `i` in the `estimate_area()` function. Is its value ever used?

(c) Explain why this version of `estimate_area()` does not work if the function `f` is not always positive.

(d) Explain what the **`if`** test does in line 13 in terms of the graphs and darts.

(e) Research the correct value of the area and compare it to values found by the simulation.

10.2 Modify Listing 10.1 so that the main program prints both the number of darts and the estimate for $n = 10, 100, 1000, \ldots 10^6$. Use a **`for`** loop. How quickly does the Monte Carlo algorithm seem to converge to the correct value?

10.3 Modify Listing 10.1 to estimate the area under $f(x) = x^3 + x^2$ between $x = -1$ and $x = 1$.

10.4 Given what you already know about computer programs, explain why it is difficult for a program to compute "random" numbers. Research the meaning of the term **pseudorandom** and report what you find.

10.5 Modify the centipede from Listing 6.1 to add randomness in some interesting way.

10.6 Write a `flip()` function that randomly returns either a 0 (representing heads) or a 1 (representing tails). Use this function to write a program `countflips.py` that asks the user for a number of trials and then simulates flipping a coin that many times. Print the number of heads and tails from the simulation. Does your program appear to be random?

10.7 Write a program `consecflips.py` that counts the number of coin flips necessary to reach a given number of heads in a row. Use the `flip()` function from Exercise 10.6. Ask the user for how many consecutive heads to wait for, and report the total number of flips needed at the end. Discuss the results of running your program.

10.8 Write a `roll()` function that simulates rolling a six-sided die by returning a random integer between 1 and 6 (inclusive). Use this function to write a program `countrolls.py` that asks the user for a number of trials

and then simulates rolling a six-sided die that many times. Print the number of times each value from 1 to 6 appears. Does your program appear to be random?

10.9 Write a program `guess.py` that asks the user for an upper limit n, and then chooses a random number between 1 and n (inclusive). Then the program asks the user for a guess and responds with either "Too high," "Too low," or "Correct." The program should continue asking for guesses until the user is correct, and then report the number of guesses made. You do not need any functions other than `main()`.

10.10 Write a program `estpi.py` that uses a Monte Carlo simulation to estimate the value of π by throwing darts at a square centered at the origin. Imagine a circle inscribed within the square, also with center at the origin. The area of the square will be easy to calculate, and the fraction of darts within the circle will allow you to estimate the area of the circle. This will give you an estimate of π. How quickly does your program converge?

10.11 A game show contestant stands in front of three large doors. Behind one of the doors is a new car; the other two doors conceal goats. The contestant chooses a door. The host then reveals a goat behind one of the other two doors, and offers the player the chance to switch to the other remaining closed door or stay with his or her original choice. Write a simulation `gameshow.py` to help you determine the best strategy for the player. Report your results.

10.12 Develop a program `coinwar.py` to simulate playing a coin-tossing version of the card game War. In this game, one player is Same and the other is Diff. Each player starts with the same number of coins. During each round, each player flips his or her coin. If the result is the same (both heads or both tails), then player Same wins both coins; otherwise, player Diff takes both coins. Play continues until one player is out of coins.

Ask the user for the starting number of coins for each player. For each round, report the flips, who won that round, and each player's new number of coins. At the end, report the winner.

This page intentionally left blank

Project: Visualization

Consider whether or not the images in the previous chapter helped you to understand Monte Carlo simulation. Would lists of numbers have been as helpful? **Visualization** has become a key application of computer technology in the recent past, thanks largely to the availability of inexpensive graphics cards. In this project, we will use turtle graphics to create an animated visualization of the Monte Carlo simulation from Chapter 10, where the dots appear randomly as they are generated by the simulation.

Chapter 6 introduced both the Python `turtle` module and the online Python documentation. The exercises will lead you to develop this visualization in stages. Use the documentation to find new turtle functions as you need them.

Exercises

1. Begin the program `visual.py` by writing a `line(x0, y0, x1, y1)` function that draws a straight line from $(x0, y0)$ to $(x1, y1)$. In `main()`, set the world coordinates to have lower-left corner $(-0.5, -0.5)$ and upper-right corner $(2, 2)$. Then use the `line()` function to draw x and y axes, as well as a horizontal line at height $y = 1$.

2. Write a `plot(f, x0, x1, n)` function to draw a plot of $y = f(x)$ over the interval $[x0, x1]$ using n line segments. Most software draws function graphs in this way, as a sequence of very short straight lines. While you could use the `line()` function from the previous exercise to do this, use the following direct algorithm instead:

   ```
   Calculate dx, the width in the x direction of each segment.
   Begin at x = x0.
   Move to (x, f(x)) without drawing.
   For each segment:
       Increase x by dx.
       Draw to the next point (x, f(x))
   ```

   Use your function to plot $f(x) = e^{-x^2}$ on $[0, 2]$. Experiment with the number of segments to get a nice image.

3. Write a `placedot(f, x, y)` function that draws the correct dot at (x, y): a large blue dot if the point is below the graph of f, or a plain dot if it is above. Use your function to finish the visualization by drawing 50 random dots within the box. Hide the turtle at the end, and close the window when the user clicks in it.

Part II

Collections and Files

This page intentionally left blank

Chapter 11

Strings

Strings are a basic data type in Python; the others you have seen so far are integer, float, and boolean. Each data type stores its information in a different way and supports a different set of operations. Whereas integers, floats, and booleans are each single pieces of data, strings provide our first example of a **data structure**, which is a **collection** of data organized to efficiently support a particular set of access methods and operations.

A Python **string** is a sequence of characters inside quotation marks, either single (') or double ("). Python also uses a form of triple quotes that you can read about in the documentation. We used strings in Listing 2.1, as well as in `print()` and `input()` expressions; now it is time to dig more deeply.

Listing 11.1: Pig Latin Translation

```
# piglatin.py

def firstvowelindex(word):
    i = 0
    while word[i] not in "aeiouy":
        i += 1
    return i

def piglatin(word):
    i = firstvowelindex(word)
    if i == 0:
        return word + "yay"
    return word[i:] + word[:i] + "ay"

def main():
    w = input("Enter a word: ")
    print("In Pig Latin, that is", piglatin(w))

main()
```

Try this program out and attempt to determine how it works. The new elements include string indexing, slicing, concatenation, and the **in** operator.

Indexing

When a string such as `lang = "python"` is stored in memory, it looks something like this, with the individual characters stored in consecutive memory locations:

`lang` ⟶

0	1	2	3	4	5
p	y	t	h	o	n

Each number along the top row is called the **index** of the character below it. Thus, 1 is the index of the `y` and the `n` has index 5. From the left end, numbering always starts at 0.

Negative indices count from the right:

`lang` ⟶

p	y	t	h	o	n
−6	−5	−4	−3	−2	−1

To access the character at index i, use the **index** operation:

`s[i]`	Character at index `i`.

For example, `lang[2]` is `"t"`, and `lang[-2]` is `"o"`. Indexing a string always returns a single character.

Slicing

Accessing a group of consecutive characters in a string is known as **slicing** in Python. It is quite powerful, as you can see from Listing 11.1.

`s[i:j]`	Slice from `i` to `j - 1`.

For example, `lang[2:4]` is `th`.

⟹ Caution: The slice `s[i:j]` does not include `s[j]`. Notice the similarity to the `range()` function.

If `i` is omitted, the slice starts from the beginning of the string; if `j` is omitted, it goes to the end. If both `i` and `j` are omitted, you get a copy of the whole string. Thus,

`s[:j]`	Slice from beginning to index `j - 1`.
`s[i:]`	Slice from `i` to the end.
`s[:]`	Full slice, which creates a **copy** of `s`.

As with `range()`, an optional third parameter `k` specifies a stepsize other than 1:

`s[i:j:k]`	Slice using stepsize `k`.

If `k` is negative, the index `i` will still be the starting point, and the slice will stop one location prior to index `j`. If `i` is omitted with a negative `k`, the beginning is considered the right end (index −1), and if `j` is omitted with a negative `k`, the end is at the left (index 0).

Concatenation

Concatenation is just a fancy word for combining strings by gluing them together one after the other. Because gluing together is a form of "adding" strings together, Python uses the addition symbol to represent concatenation:

`s + t`	The string `s` followed by `t`.
`s += t`	Shorthand for `s = s + t`.

Thus, `"abc" + "def"` is `"abcdef"`. Multiplication is repeated addition, and so we also have a repeated concatenation operator:

`s * n`	The string `s + s + s + ... + s`, `n` times.
`n * s`	Same as `s * n`.

In and Not In

To test whether or not a character is a vowel in Listing 11.1, we used the **`in`** operator:

`x` **`in`** `s`	`True` if `x` is a substring of `s`; otherwise, `False`.
`x` **`not in`** `s`	Opposite of **`in`**.

Being a **substring** means that the characters of `x` occur as a consecutive slice of `s`. For example, `"e"` and `"iou"` are substrings of `"aeiouy"`, but `"w"` and `"eou"` are not.

Exercises

11.1 Give the data type returned by the **`in`** and **`not in`** operators.

11.2 Suppose `word = "image"` and `phrase = "protein synthesis"`.

(a) Determine the values of these Python expressions:

i. `word[0]`
ii. `word[2:]`
iii. `word[:2]`
iv. `word[::2]`
v. `phrase[1:5]`
vi. `phrase[:-4]`
vii. `phrase[:8] + word`

(b) Create Python expressions using `word` and `phrase` as above that have these values:

i. `"g"`
ii. `"mag"`
iii. `"pro"`
iv. `"thesis"`
v. `"synth"`
vi. `"image synthesis"`

11.3 Describe in words what these slices do:

(a) `s[::2]`

(b) `s[::-1]`

11.4 Determine a slice for each of these tasks:

(a) Remove the last character of a string `s`.

(b) Duplicate a string `s`.

11.5 Write an **`if`** statement that returns `False` if the first character of the string `s` is a digit. This test would be useful in a function that tests for legal Python identifiers.

11.6 Use Listing 11.1 to answer these questions:

(a) Trace the execution of the program with input "spy."

(b) Trace the execution of the program with input "nix."

(c) Identify the accumulator in the program and describe how it is being used.

(d) Explain why no **`else`** clause is necessary for the **`if`** at line 11.

11.7 Different people use different rules for Pig Latin. Modify Listing 11.1 to use whatever system you learned. Explain the differences between your rules and those of Listing 11.1.

11.8 Modify Listing 11.1 to insert a hyphen in the Pig Latin form: e.g., for the word "hello," return "ello-hay."

11.9 Describe what Listing 11.1 does with words that begin with a "y," such as "yellow." Modify the program to work correctly on those inputs.

11.10 Describe what Listing 11.1 does with words that contain no vowels, such as "nth." It's not obvious what to do in that case, but modify the program to do something sensible. You may need the `len()` function described in Chapter 14.

11.11 Write a program `username.py` that builds a computer system username given a person's name. Ask the user for the first name and last name (separately), and then display the username, which (for the purposes of this exercise) is at most 12 characters of the last name, followed by the first initial of the first name, followed by a "1." Write a function to return the username.

Chapter 12

Building Strings

Strings are useful for more than representing sentences and words. For example, genetic information is usually represented by character strings.

DNA is a double helix of two chains of nucleotides. Each nucleotide base can be represented by a single letter, and so a chain of nucleotides can be thought of as a string. Even though DNA has two chains (also called strands), the two are closely related: given one, it is easy to calculate the other. Thus, DNA is generally described by one string of characters representing the nucleotide bases of one of its strands.

The four bases that make up DNA are adenine (A), cytosine (C), guanine (G), and thymine (T).

The second strand of DNA is always the **reverse complement** of the first. The **complement** of a strand of DNA swaps each base with its complementary base: $A \leftrightarrow T$ and $C \leftrightarrow G$, and the reverse complement just reverses the order of the complementary sequence. For example, the complement of `AGGTC` is `TCCAG`, and the reverse complement is `GACCT`.

Listing 12.1: DNA Sequences

```
# dna.py

from random import choice

def complement(dna):
    result = ""
    for c in dna:
        if c == "A":
            result += "T"
        elif c == "T":
            result += "A"
        elif c == "C":
            result += "G"
        elif c == "G":
            result += "C"
    return result

```

```
18 def reversecomp(dna):
19     return complement(dna)[::-1]
20
21 def random_dna(length=30):
22     fragment = ""
23     for j in range(length):
24         fragment += choice("ACGT")
25     return fragment
26
27 def main():
28     for i in range(10):
29         dna = random_dna()
30         print(dna + "   " + reversecomp(dna))
31         print(complement(dna) + "\n")
32
33 main()
```

This program generates random strings of DNA and displays each with its reverse complement. On the left side of the output, you will see the random strand with its complement below it. This is how the two strands are tied together in the double-helix: each base binds with its complement across the helix. However, while the top strand is read left-to-right, the second strand is read from right to left, and so it is the reverse complement, printed on the right, that is more useful than the complement.

String Accumulators

Both the `complement()` and `random_dna()` functions use **string accumulation loops** to build their return values. These follow the same pattern as numeric accumulation loops (see Chapter 4):

```
<accumulator> = <starting value>
loop:
    <accumulator> += <string to add>   # adds on the right
```

The reason these are so similar is that concatenation is analogous to addition for strings, and Python uses "+" to represent both.

There are two main differences between numeric and string accumulators:

The starting value for string accumulators is usually the **empty string** `""`, denoted in Python by two quotation marks with nothing between them.

Concatenation is not commutative. Adding on the right is usually different from adding on the left (see Exercise 12.2):

`s = s + t`	Add `t` to `s` on the right.
`s = t + s`	Add `t` to `s` on the left.

Most of the time (as in Listing 12.1), you will want to add on the right, and you can use the shorthand "`+=`" to do that. However, if you need to add on the left, you will have to write out the full statement instead of using the shorthand.

Loops over Strings

There is another new feature in the `complement()` function of Listing 12.1: the **`for`** loop in line 7. In fact, the syntax is not new:

```
for <variable> in <string>:  # loop over each character
    <body>
```

Compare this with the syntax given in Chapter 4: they are identical except that the earlier version wrote `<sequence>` instead of `string`. Python treats strings as sequences of characters, and so when a string is used in a **`for`** loop, the variable takes on the value of each character in the string.

Escape Sequences

The `main()` function in Listing 12.1 uses one other new feature: an **escape sequence** inside of a string. Escape sequences begin with a **backslash** "\" and are used to insert non-alphabetic characters into a string:

`\n`	Newline.
`\t`	Tab.
`\"`	To get " inside a double-quoted string.
`\'`	To get ' inside a single-quoted string.
`\\`	If you need a backslash itself.

Escape sequences and concatenation give us more control over printing than we have had to this point.

Exercises

12.1 Use Listing 12.1 to answer these questions:

(a) Identify the accumulator variables in both the `complement()` and `random_dna()` functions. Do these accumulate on the right or left? How do you know?

(b) Find the code that reverses the (complement) string.

(c) Modify the `reversecomp()` function to call a separate `reverse(s)` function to reverse the complement. Write the `reverse()` function, as well.

(d) Look up the `choice()` function from the `random` module and explain what it does in the `random_dna()` function.

12.2 (a) Give example strings to show that concatenation on the right may produce different results from concatenation on the left.

(b) Give example strings to show that concatenation on the right may produce the same result as concatenation on the left.

12.3 Use a string accumulator to write a `reverse(s)` function that returns the string `s` in reverse order. Do not use a slice. Modify Listing 12.1 to use your function.

12.4 Use a string loop to write a function `is_dna(s)` that returns `True` if `s` is a string of DNA nucleotides and otherwise returns `False`. Test your function on random strings of DNA, as well as other, non-DNA strings.

12.5 Write a function `random_rna(length)` that returns a random fragment of RNA of the given length. **RNA** uses the same bases as DNA, except that uracil (U) replaces thymine (T). Include a `main()` function that prints 10 random strings of RNA of length 30.

12.6 Write a function `is_rna(s)` that returns `True` if `s` is a string of RNA nucleotides and otherwise returns `False`. Test your function on random strings of RNA and DNA.

12.7 DNA **transcription** transforms a strand of DNA to RNA by replacing every thymine (T) with uracil (U). Write a function `transcription(dna)` that takes a DNA string and returns the corresponding RNA. Use a string accumulator. Test your function on random strings of DNA.

12.8 A fragment of DNA is a **palindrome** if it is the same as its reverse complement. (Note that this is different from the definition of English palindromes.) Write a function `ispalindrome(dna)` that returns `True` if the given `dna` is palindromic, and otherwise returns `False`. Use your function to write a program that finds a palindrome of length 10 by testing randomly generated strings of DNA until it finds one.

12.9 Write a function `countbases(dna)` that counts the number of each of the four bases in the given string of `dna`. Return all four counts, separated by commas. Test your function on random strings of DNA.

12.10 Write a function `dec_to_bin(n)` that takes a nonnegative integer `n` and returns the corresponding binary string (without the `"0b"` prefix). Do not use the built-in `bin()` function. Instead, use a string accumulator:

```
repeat while n is positive:
    concatenate the bit n % 2 to the left end of the result
    integer-divide n by 2
```

You may need a type conversion from Chapter 3. Write a `main()` that tests your function on all integers between 0 and 100.

12.11 Write a function `bin_to_dec(s)` that takes a binary string (without the `"0b"`) and returns the corresponding decimal integer. You may need a type conversion from Chapter 3. Test your function on the output from the previous exercise.

This page intentionally left blank

Project: ISBN Check Digits

Error detection and correction are important requirements for reliable electronic communication. A common error detection technique is to append a **check digit** to the end of a piece of data, where the check digit is calculated from that data. After receiving transmission of data with a check digit, the receiver can perform the same calculation, and if the check digits do not match, then there must have been an error and the data can be resent. Check digits are used in UPC bar codes, credit card numbers, and ISBNs.

The check digit of a 10-digit ISBN is calculated from the first nine digits d_0, d_1, ..., d_8 as

$$(1d_0 + 2d_1 + 3d_2 + \cdots + 9d_8) \bmod 11.$$

The check digit is then used as the tenth digit of the ISBN. Because mod 11 values may be as large as 10, the letter "X" (Roman numeral ten) is used as the check digit if the result is ten.

Conditional Expressions

Fairly often, you will find yourself wanting to set a variable or return a value based on a boolean test:

```
if <boolean>:
    <var> = <expr1>
else:
    <var> = <expr2>
```

```
if <boolean>:
    return <expr1>
else:
    return <expr2>
```

Python offers a shorthand **conditional expression** syntax that you may find preferable:

```
<expr1> if <boolean> else <expr2>
```

The value of the above expression is `<expr1>` if `<boolean>` is `True`; otherwise, it is `<expr2>`. The two forms above are rewritten like this to use conditional expressions:

```
<var> = <expr1> if <boolean> else <expr2>
```

and

```
return <expr1> if <boolean> else <expr2>
```

Exercises

1. If the check digit calculation performed by the receiver is correct (i.e., it matches the digit that was sent), is the data transmission guaranteed to be correct? Explain why or why not.

2. List some possible reasons that the ISBN check digit calculation might use the multiplicative coefficients 1, 2, ..., 9 instead of just adding up the digits. What types of errors might the coefficients help detect?

3. Write a function `check10(s)` that returns the check digit of a 9-digit ISBN string `s` without hyphens. Use a conditional expression. Write a `main()` that asks the user for input.

4. Research the new standard ISBN-13 and write a function `check13(s)` to compute the check digit for it. Use a conditional expression.

Chapter 13

Computer Memory: Text

Recall from Chapters 5 and 9 that both integers and floating-point numbers are stored in binary, although the exact format for each is different. Binary is used because computer memory is essentially a sequence of on/off switches, and these can easily be thought of as 0's and 1's.

This raises the question: how is text stored in memory? The basic idea is simply to assign a code number to each character. Then, for each character, we just store the corresponding code number in binary.

The standard coding system for the English alphabet is **ASCII**, pronounced "ask-ee." Standard ASCII codes are seven bits long, although each character usually occupies eight bits because that makes a complete byte. There has been disagreement about how to use that last eighth bit; however, more recently, Unicode is becoming the standard for international communication. It is a two-byte code that extends ASCII.

One of the features of ASCII is that digits and letters appear sequentially:

0–9	Codes 48–57
A–Z	Codes 65–90
a–z	Codes 97–122

Python offers two built-in functions that allow you to work directly with ASCII codes:

`chr(n)`	Character with ASCII code `n`.
`ord(c)`	ASCII code for the character `c`.

Exercises

13.1 Give the range of codes available for seven-bit ASCII if stored as unsigned integers.

13.2 Give the range of codes available for eight-bit extended versions of ASCII.

13.3 Look at binary representations of the ASCII codes for several corresponding pairs of upper and lowercase letters (for example, "E" and "e"). Find the pattern, and then use it to explain why the lowercase group does not immediately follow the uppercase group.

13.4 Explain why curly brackets `{}` occur at the end of the list of symbols in the index of this text. Where would they be listed relative to alphabetic characters (a–z and A–Z) if the symbols were not listed separately?

13.5 Write a program `ascii.py` to display an ASCII table for the characters with codes between 32 and 126.

13.6 Write the Python function `encode(msg)` that returns a string containing the ASCII codes of each character in `msg`. For example, `encode("ABC")` should return the string `"65 66 67"`.

Chapter 14

Lists

The **sieve of Eratosthenes** is an algorithm for making a list of primes that dates back to at least the third century BC:

```
List the integers, beginning with 2.
Circle 2 (it must be prime),
    and cross out all of its multiples.  They cannot be prime.
Circle the next integer that is not yet crossed out (3),
    and cross out its multiples.
Repeat.
```

Here is a straight-forward implementation that creates a list of the primes. Recall that the output of the `range()` function is not itself a list but may be converted to one.

Listing 14.1: Sieve of Eratosthenes

```
# sieve.py

def sieve(n):
    primes = list(range(2, n + 1))
    for k in range(2, n + 1):
        if k in primes:
            for j in range(2 * k, n + 1, k):
                if j in primes:
                    primes.remove(j)
    return primes

def main():
    n = int(input("Enter upper limit: "))
    print("The primes up to", n, "are:\n", sieve(n))

main()
```

This program contains nesting that is about as deep as you ever want to go: there is an **if** inside of a **for** inside an **if** inside of a **for** loop.

Lists

Given our description of the sieve, a list should seem like a natural choice to use for its implementation. You are already somewhat familiar with lists, for two reasons. First, we discussed lists briefly in Chapter 4, and second, lists are very similar to strings.

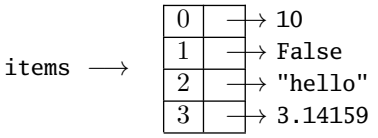
A Python **list** stores an ordered sequence of items. But while every item in a string is a single character, list elements may be of any Python type. For example, this list:

```
items = [10, False, "hello", 3.14159]
```

contains an integer, a boolean, a string, and a float. Lists are written inside square brackets. A pair of empty brackets `[]` denotes the **empty list**.

Lists Are Like Strings

Lists are stored in memory exactly like strings, except that because some of their objects may be larger than others, they store a reference at each index instead of a single character. A **reference** in Python is just a memory location that points to where an object is actually stored. (Thus, references are also called **pointers**.) For example, the list named `items` given above will be stored in memory something like this:



Each of the individual items in the list is stored somewhere else in memory.

In fact, *all* Python variables are references, but we have ignored that detail to keep things simpler.

Because lists and strings share this similar structure, the following operations all work the same for lists as they do for strings:

- Indexing using `[]`
- Slicing using `[i:j:k]` (including full slice copies)
- Concatenation using `+`
- Repeated concatenation using `* n`
- **`for`** loops
- Accumulation loops
- **`in`** and **`not in`**

⟹ Caution: Concatenation only works between objects of the same type. For example,

```
items = [1, 4, 7] + "abc"
```

will cause an error because the type of `[1, 4, 7]` (list) doesn't match the type of `"abc"` (string).

Several other built-in functions are available for both lists and strings:

`len(x)`	Number of elements in `x`.
`min(x)`	Smallest element in `x`.
`max(x)`	Largest element in `x`.

Finally, recall from Chapter 4 that there is a type conversion function for lists:

`list(x)`	Convert `x` to a list.

Lists Are Also Not Like Strings

These operations work with lists but not strings:

`items[i] = x`	Replace `items[i]` with `x`.
`items[i:j] = newitems`	Replace items in slice with `newitems`.
`items[i:j:k] = newitems`	Replace items in slice with `newitems`.
`del` `items[i]`	Remove `items[i]`.
`del` `items[i:j]`	Remove items in slice.
`del` `items[i:j:k]`	Remove items in slice.
`sum(items)`	Sum of the elements in `items`.

Except for `sum()`, these all modify the original list.

⟹ Caution: Never modify a list that is driving a **`for`** loop. For example, the loop at line 5 of Listing 14.1 cannot be over the list `primes`, because that list changes inside the body of the loop.

Random Functions for Lists

The `random` module includes these functions for lists:

`choice(items)`	One random element from the `items`.
`shuffle(items)`	Randomly shuffle the elements in `items`.

The `choice()` function also works on strings; the `shuffle()` function does not.

Objects and Object Methods

There is one more piece of new syntax in Listing 14.1 that is going to take some background explanation. Every data type in Python is what is known as an object data type, and in order to understand some of the functionality of Python lists (and strings), we need to begin to use object terminology.

For our purposes, an **object** data type is one that not only describes how data will be stored and what operations can be done with it; it also provides additional functionality via **methods**, which are specialized functions that you can ask the object itself to perform. Programming that focuses on using object data types is referred to as **object-oriented**. The programming you have done to this point without thinking in terms of objects is usually called **imperative** or **procedural**.

Method Calls

The syntax to call a method from an object is called **dot notation**:

```
<object>.<method>(<arguments>)
```

Compare this with the syntax for a function call in Chapter 3. The differences are that you must ask a particular object to perform a method, and there is a dot (period) between the object and the name of the method. Calling the same method from different objects will usually produce different results.

List Methods

If `items` is a list object, these are some of the methods that may be called on it:

`items.append(x)`	Add item `x` to the end of `items`.
`items.insert(i, x)`	Insert item `x` into `items` at index `i`.
`items.pop()`	Remove and return the last item in `items`.
`items.pop(i)`	Remove and return `items[i]`.
`items.remove(x)`	Remove item `x` from `items`. Raises `ValueError` when `x` not in `items`.
`items.reverse()`	Reverse the order of the elements in `items`.
`items.sort()`	Sort the list `items`.

The `remove()` method raises an **exception** if the requested item is not in the list. Exceptions are a Python mechanism for handling errors; for now, we will just try to avoid them.

$\Longrightarrow$ Caution: These methods all modify the list they are called on, and only `.pop()` returns anything. All others return the Python object `None`, which is returned by any function that does not specify a return value.

Exercises

14.1 Describe how to spot method calls in Python code.

14.2 Perform the sieve on paper for integers up to 100. Give the resulting list of primes.

14.3 Trace the execution of `sieve(10)` for $k = 2$ and $k = 3$. Show the values of `j` and the contents of the list `primes` at the end of each execution of the `k`-loop.

14.4 Use Listing 14.1 to answer these questions:

(a) Identify the method call, along with the name of the object, the name of the method, and any arguments.

(b) Explain why `n + 1` is used in three places inside the `sieve()` function instead of `n`.

(c) It looks as if lines 5 and 6 could be combined into one statement more efficiently as "`for k in primes:`". Does this change work? Explain why or why not.

(d) Explain the role of the step size `k` in line 7.

(e) Explain the need for the `if` statement in line 8.

14.5 Explain why the `k` loop in the `sieve()` function could stop at $k = \sqrt{n}$ instead of $k = n$ and still work correctly. Modify Listing 14.1 to reflect this improvement. You may want to use the `ceil()` function from the `math` module.

14.6 Explain why the `j` loop in the `sieve()` function could start at $j = k^2$ instead of $j = 2k$, and still work correctly. Modify Listing 14.1 to reflect this improvement.

14.7 Write a function `randints(n, a, b)` that returns a list of `n` random integers between `a` and `b` (inclusive). The integers do not need to be distinct. Use a list accumulator to build the list. Write a `main()` to test your function.

14.8 Write a function `randintsdistinct(n, a, b)` that returns a list of `n` distinct random integers between `a` and `b` (inclusive). Write a `main()` to test your function.

14.9 Write a function `randfloats(n, a, b)` that returns a list of `n` random floats from the interval $[a, b]$. Write a `main()` to test your function.

14.10 Write a `mysum(items)` function that returns the sum of the elements in the list `items` without using the built-in `sum()` function. Test your function on random lists against the built-in function.

14.11 Write a `mymin(items)` function that returns the smallest item in the list `items` without using the built-in `min()` function. Test your function on a random list of numbers, a list of strings, and a string.

14.12 Write a `mymax(items)` function that returns the largest item in the list `items` without using the built-in `max()` function. Test your function on a random list of numbers, a list of strings, and a string.

14.13 Write a `mean(items)` function that returns the average of the values in the list `items`. Assume all entries are numeric. Test your function on random lists.

14.14 Write a `median(items)` function that returns the median of the values in the list `items`. Look up the definition of median if you need it, and test your function on random lists. Do not modify the original list; instead, make a copy of the list before sorting it. Test your function on random lists.

14.15 Write a `evens(items)` function that returns a new list of only the even integers in the list `items`. Test your function on random lists.

14.16 Write a `odds(items)` function that returns a new list of only the odd integers in the list `items`. Test your function on random lists.

14.17 Write a `block(names, blocked)` function that returns a new list of only the names in the list `names` that are not in the list `blocked`. Write a `main()` to test your function.

14.18 Write a `mychoice(items)` function that returns a random element from the list `items` without using the `choice()` function. You may use other functions from the `random` module. Write a `main()` to test your function.

14.19 Write a function `delrand(items)` that deletes and returns one random element from the list `items`. Write a `main()` to test your function.

14.20 Improve Exercise 10.6 by modifying the `flip()` function to randomly return either the string `"heads"` or the string `"tails"`.

14.21 Use a list to improve the `countrolls.py` program from Exercise 10.8. Hint: start with a list of 0's.

14.22 Write a program `randmadlib.py` that modifies Listing 2.1 to generate random Mad Libs (in correct grammatical form) by choosing random nouns, verbs, adjectives, etc., instead of asking the user for them. Make your own lists of each type of word and create your own sentence template.

14.23 Write a function `swap(items, i, j)` that swaps the elements `items[i]` and `items[j]`.

14.24 Write a function `myreverse(items)` that reverses the list `items` without using the list method `.reverse()`. Do not return anything; just change the order of the elements in the given list. Test your function on random lists.

14.25 Write a function `myshuffle(items)` that shuffles the list `items` without using the `shuffle()` function from the `random` module. This is trickier than it sounds, but here is an algorithm that works, where n is the length of the list:

```
for i = 0 to n - 2:
  choose a random j between i and n - 1 (inclusive)
  swap items i and j
```

Explain why the **for** loop does not need to go to $n - 1$. Write a `main()` to test your function.

14.26 Write a function `issorted(items)` that returns `True` if the list `items` is sorted in increasing order; otherwise, it should return `False`. Hint: it is enough to compare each element with the one next to it.

14.27 Write a function `mysort(items)` that sorts the given list without using the `.sort()` method. Here is an algorithm that will work:

```
Find the smallest element and swap it with item 0.
Find the next smallest element and swap it with item 1.
Continue.
```

Test your function on known lists and random lists.

14.28 Write a function `primefactors(n)` that returns a list of the (unique) prime factors of the integer `n`. For example,

```
primefactors(24) = [2, 3]
```

Test your function on $n = 2$ through 100.

14.29 Write a function `primefactorization(n)` that returns a list that represents a complete factorization of `n` into primes. For example,

```
primefactorization(24) = [2, 2, 2, 3]
```

Test your function on $n = 2$ through 100. Hint: think of repeated use of `smallestdivisor()`.

14.30 Write a function `divisors(n)` that returns a list of the proper positive divisors of the integer `n`, including 1 but excluding `n` itself.

14.31 Write a function `isperfect(n)` that returns `True` if `n` is a **perfect number**, meaning it is positive and equal to the sum of its proper divisors (see the previous exercise). Use your function to find all perfect numbers less than 10,000.

14.32 The **Fibonacci sequence** is

$$1, 1, 2, 3, 5, 8, 13, 21, 34, 55, \ldots$$

defined by beginning with two 1's and then adding the two previous numbers to get the next one. Write a function `fib(n)` that returns a list of the first `n` Fibonacci numbers. How large is the 100th Fibonacci number?

Project: Program Performance

Exercises 14.5 and 14.6 describe two of the improvements that can be made to the efficiency of Listing 14.1. Because they reduce the number of loop executions, they must make the program faster, but by how much? Is the improvement significant? Measuring **program performance** is one way to analyze the impact of changes made to your code.

In Python, we can get reasonably accurate timings using this function from the `time` module:

`clock()`	Current processor time.

The standard way to use a clock is to surround the code you want to measure with two calls to the clock, like this:

```
start = time.clock()
sieve(10000)
stop = time.clock()
elapsed = stop - start
```

The variable `elapsed` will then hold the time it took to execute the function. Be sure not to put any other code between the two calls to the clock.

Exercises

1. Discuss the difficulties in accurately measuring a program's execution time. What other factors might influence the elapsed time?

2. Write a function `primelist(n)` that uses `isprime()` from Listing 8.1 instead of the sieve to return a list of primes less than or equal to `n`. Test that this function returns the same list as `sieve()`.

3. A list of boolean values can be used to implement the sieve rather than a list of integers. If the name of the boolean list is `primes`, then the idea is that

`primes[i] = True`	If `i` is not crossed off (potential prime).
`primes[i] = False`	If `i` is crossed off (not prime).

Write a function `sieve2(n)` to compute and return the boolean list that results from using this technique to implement the sieve.

4. Write a function `test(plist, blist, n)` that returns `True` if the list of integer primes less than or equal to `n` in `plist` is the same as that represented by the boolean list `blist`. Return `False` otherwise.

5. Measure the performance of these approaches to finding a list of primes for $n = 20000$:

   ```
   sieve(n)
   primelist(n)
   sieve2(n)
   ```

   Report and discuss your results.

Project: Heat Diffusion

Given an insulated cylindrical metal rod with a heat source at each end, the temperature of the rod will eventually stabilize so that it varies linearly between the temperatures of the two heat sources. The physics and mathematics describing this process are beyond the scope of this course (see, e.g., [2]), but we will be able to simplify the mathematics and use Python lists to simulate the diffusion of heat in the rod.

The key idea in translating from the physical situation to a program is to **discretize** the problem, meaning, in this case, that we break up the continuous rod into a discrete set of short pieces. That allows us to consider all of the heat transfer happening at the boundaries between the pieces. If we keep track of the temperature at each of these boundaries, we have a list. For example, if the rod begins at $0°C$, the ends are heated to $30°C$ and $50°C$, and we imagine ten pieces, then the list

```
temps = [30, 0, 0, 0, 0, 0, 0, 0, 0, 0, 50]
```

represents the initial state of the bar at the 11 boundaries. Our program will update these temperatures over time to simulate the diffusion of heat.

The program will also discretize time, breaking it into small pieces of length dt. Remarkably, if we choose dt carefully, a reasonably accurate algorithm to simulate diffusion is simple: to calculate the temperature of a boundary at the next time step, we just average the temperatures to its left and right.

Listing 14.2: Heat Diffusion (Algorithm)

```
Set initial temps.
Repeat:
    for each internal boundary temp (not the ends):
        new temp = average old temp to left and old temp to right
    copy new temps to old
```

$\Longrightarrow$ Caution: It is important to calculate all the new temperatures based on the old temperatures. That is why there is what seems like an extra step of copying the new temperatures to the old locations. Remember that a full slice (`[:]`) will copy a list.

We calculate dt from the width h of each piece:

$$dt = \frac{1}{2}h^2$$

Exercises

1. Write a function `init(left, right, inittemp, n)` that returns a list of the initial boundary temperatures, given the fixed left and right temperatures, the initial temperature of the rod itself, and the number of segments `n`.

2. Write a program `diffusion.py` to perform the simulation. Because it is helpful to see the values of the temperatures at each step of the simulation, write the rest of your code in `main()`. (We generally try to avoid print statements in any functions other than `main()`.) Use these values, in addition to the initial temperature conditions given above:

Rod length	5 cm
Number of segments	10
Duration of the simulation	1 sec

   Determine the temperature at the center of the bar at the end of the simulation.

Chapter 15

Files

Working with large blocks of text in a file opens up a whole new range of possibilities for interesting programs. The following program solves Jumble puzzles, which ask the solver to figure out an unknown word given a scrambled set of its letters.

One of the difficulties in solving this problem is that long words have so many different reorderings, called **permutations**. The strategy employed here is to create a unique signature for each word, so that a word has the same signature as all of its permutations. This program also finds **anagrams**, since typing in an unscrambled word will find all of its anagrams.

Listing 15.1: Solve Jumble

```
1  # solvejumble.py
2
3  def signature(word):
4      chars = list(word)
5      chars.sort()
6      return chars
7
8  def wordlist(fname):
9      with open(fname) as f:
10         return f.read().split()
11
12 def matches(jumble):
13     sign = signature(jumble)
14     result = []
15     for word in wordlist("dictionary.txt"):
16         if signature(word) == sign:
17             result.append(word)
18     return result
19
20 def main():
21     jumble = input("Enter a scrambled word: ")
22     print("Matches are:", matches(jumble))
23
24 main()
```

Files

Python makes it particularly easy to access files. A **file** is a collection of data that is stored by an operating system in such a way that it can be retrieved by its name and location in the directory or folder hierarchy. The operating system hides all details about exactly where the bytes that make up the file are located on the hard disk or other storage medium.

The built-in function `open()` gives access to files in Python:

`open(fname)`	Open file named `fname` in current directory.
`open(fname, "r")`	Open file for reading (the default).
`open(fname, "w")`	Open file for writing.

Each of these returns a **file object**, which you can call file methods on (listed below). Files opened as above will be in **text mode**. Specifying `"rb"` or `"wb"` opens the file in **binary mode**, meaning that bytes in the file are read "as-is," without being interpreted as characters.

Because things can go wrong when opening a file—for example, it may be corrupted or not exist—it is important to open files carefully so that errors (raised as exceptions) may be handled gracefully.

With Statements

The recommended way to open a file in Python is to use a **`with`** statement:

```
with <expression> as <variable>:
    <body>
```

Inside the `<body>` of a **`with`** statement, the `<variable>` takes on the value of the `<expression>`. However, much more happens behind the scenes. In this case, the **`with`** statement in line 9 makes sure that the file is closed properly after execution of the `<body>`, even if an exception occurs.

File Methods

Here are some of the methods that may be used with a file object named `f`:

`f.read()`	Entire contents of `f` as a string.
`f.readline()`	Next line from `f`, including the trailing newline. Returns empty string at end of file.
`f.readlines()`	List containing all lines from `f`.
`f.write(s)`	Write string `s` to `f`.
`f.close()`	Close `f`. Called automatically by **`with`**.

The line-based methods may only be used with text files.

$\Longrightarrow$ Caution: Different operating systems use different escape sequences for the newlines at the end of each line of a text file, called trailing newlines:

Operating System	**Escape Sequence**	**ASCII (decimal)**
Unix® family (Linux®, Mac OS® X)	`\n`	10
Microsoft Windows	`\r\n`	13, 10
Mac OS 9 and earlier	`\r`	13

Fortunately, Python automatically translates newlines so that in code you can always just use `\n`.

File Loops

If you need to loop over all the lines in a file, Python allows you to put the file object directly in a **for** loop:

```
for <variable> in <file>:    # loop over each line in the file
    <body>
```

This is essentially equivalent to:

```
for <variable> in <file>.readlines():
    <body>
```

Therefore, if you use a **for** loop, each line will contain its newline escape sequence.

Splitting Strings

Strings are also objects in Python, and one of their most useful methods in conjunction with reading files is `.split()`:

`s.split()`	List of "words" in string `s`.

"Words" in this case are defined as any groups of characters separated by **whitespace**, which is made up of spaces, tabs, and newlines. For example,

```
"Hello, world.".split() = ["Hello,", "world."]
```

Notice that after a `.split()`, punctuation attaches to the word preceding it.

We will explore other string methods in the next chapter.

Multiple Method Calls

It is common in Python to call methods one right after another. For example, consider line 10 of Listing 15.1:

```
return f.read().split()
```

Read the method calls left to right: the result of `f.read()` is a string, and then the `.split()` method is called on that string.

Exercises

15.1 Use Listing 15.1 to answer these questions:

(a) Identify the accumulation loop in Listing 15.1, along with the type of accumulator. Explain how you know the type.

(b) What type of object is the loop in line 15 over? Explain how you know.

(c) Rewrite the multiple method call in line 10 to use two separate steps.

15.2 Answer these questions about the `signature()` function in Listing 15.1:

(a) Give the signature that is calculated for each of these words: file, string, python.

(b) Does it work to replace lines 4 and 5 with this one method call:

```
chars = word.sort()
```

Explain why or why not.

(c) Does it work to combine lines 4 and 5 into a multiple method call like this:

```
chars = list(word).sort()
```

Explain why or why not.

(d) Does it work to replace lines 5 and 6 with

```
return chars.sort()
```

Explain why or why not.

(e) Does it work to replace the entire function body (lines 4–6) with

```
return list(word).sort()
```

Explain why or why not.

15.3 Write a program `printfile.py` that asks for the name of a file and then prints the contents of that file. Write it in four different ways:

(a) Using `read()`

(b) Using `readlines()`

(c) Using `readline()`

(d) Using a file **`for`** loop

Which of these methods appears double-spaced, and which do not?

15.4 Fix the double-spacing in the previous exercise by:

(a) Using the `end=` option with your `print()` statements

(b) Using a slice to remove the trailing newlines

15.5 Write a program `reverse.py` that prints the lines of a file in reverse order.

15.6 Write a program `wotd.py` that prints a random word of the day from a dictionary.

15.7 Write a program `wc.py` that works like the Unix `wc` program: given a file name, it prints the number of lines, number of words, and number of characters in the file. Ask the user for the file name.

15.8 Write a program `cp.py` that works like the Unix `cp` command: given two file names, it copies the first file (source) to the second (destination). Ask the user for the names of the source and destination.

15.9 Write a program `avgword.py` that computes the average word length for all the words in a file. Do not worry about punctuation, and ask the user for the file name.

15.10 Write a program `linenum.py` that asks for the name of a file and then prints the contents of the file with line numbers along the left edge.

15.11 Write the Python function `decode(msg)` that undoes the encoding from Exercise 13.6.

15.12 Modify Listing 15.1 to find the words that have the most anagrams. Warning: this program will take a *long* time to run. However, see Exercise 18.11.

15.13 Assign each letter a value based on its order in the alphabet, so that $a = 1$, $b = 2$, $c = 3$, and so on. A word's value is then the sum of the values of its letters. Write a program to find the word with the highest value.

This page intentionally left blank

Chapter 16

String Methods

Files are not limited to those on your hard drive. The following program retrieves a web page from the Internet, and then uses Python's string methods to display specific information from it.

Listing 16.1: GCS Menu

```
# menu.py

import urllib.request

URL = "http://www.central.edu/go/gcsmenu"

def getpage(url):
    with urllib.request.urlopen(url) as f:
        return str(f.read())

def gettag(page, tag, start=0):
    opentag = "<" + tag + ">"
    closetag = "</" + tag + ">"
    i = page.find(opentag, start)
    if i == -1:
        return None, i
    j = page.find(closetag, i)
    return page[i + len(opentag):j], j

def process(page):
    heading, i = gettag(page, "h2")
    result = "\n" + heading.center(60) + "\n\n"
    day, i = gettag(page, "h3", i)
    while day is not None:
        result += day + "\n"
        meal, i = gettag(page, "p", i)
        result += "  " + meal.strip("<>p") + "\n"
        day, i = gettag(page, "h3", i)
    return result

def main():
```

```
32      page = getpage(URL)
33      print(process(page))
34
35  main()
```

Visit the URL in a web browser and use "View Page Source" to see the raw contents of the page. Listing 16.1 searches for data contained in **HTML tags** such as `<h2>Grand Central Station Menu</h2>`.

In addition to the new string methods, this example uses the **`is`** comparison, multiple assignment and return, constants, a different version of the **`import`** statement, and the `urllib.request` module.

String Methods

Python provides many string methods. A few are highlighted here, but see the documentation for a complete list. These search within a string `s`:

`s.find(t[, start[, end]])`	First index where `t` is a substring in `s[start:end]`.
`s.rfind(t[, start[, end]])`	Last index where `t` is a substring in `s[start:end]`.
Return −1 if not found. Optional `start`, `end` work like slices.	
`s.startswith(t)`	`True` if `t` is a prefix of `s`.
`s.endswith(t)`	`True` if `t` is a suffix of `s`.
`s.count(t)`	Number of occurrences of substring `t` in `s`.

Square brackets in syntax descriptions, such as those used above, indicate optional elements. With the `.find()` methods, because there are two sets of brackets, there are actually three options:

```
s.find(t)          # Search s
s.find(t, i)       # Search s[i]
s.find(t, i, j)    # Search s[i:j]
```

These test the contents of `s`:

`s.isalpha()`	`True` if all characters in `s` are alphabetic.
`s.isupper()`	`True` if all characters in `s` are uppercase.
`s.islower()`	`True` if all characters in `s` are lowercase.
`s.isdigit()`	`True` if all characters in `s` are digits.

These each return a modified copy of `s`:

`s.upper()`	All uppercase.
`s.lower()`	All lowercase.
`s.capitalize()`	First letter capitalized and the rest lowercase.
`s.title()`	Each word capitalized, rest lowercase.
`s.replace(old, new)`	All occurrences of substring `old` replaced by `new`.
`s.center(width)`	Centered in a string of width `width`.
`s.strip([chars])`	All `chars` removed from both ends.
`s.lstrip([chars])`	All `chars` removed from left end.
`s.rstrip([chars])`	All `chars` removed from right end.
If optional `chars` *omitted, whitespace is removed.*	

$\Longrightarrow$ Caution: None of these methods changes the original string; they modify and return a copy.

The `is` Comparison

Recall that Python has the special value `None` to represent "nothing" or "no object." The `gettag()` function in Listing 16.1 uses `None` to indicate that a tag was not found. The proper way to test for `None` in the calling function on line 24 is with an **`is`** comparison:

`obj1` **`is`** `obj2`	`True` if `obj1` and `obj2` refer to the same object.
`obj1` **`is not`** `obj2`	Opposite of **`is`**.

The **`is`** comparison never checks the value of its object references; it only checks whether or not the references refer to precisely the same object.

Multiple Assignment and Return

Python allows you to assign values to more than one variable at a time, as long as the number of values on the right side of the equals sign is the same as the number of variables on the left:

```
<var1>, <var2>, ..., <varN> = <expr1>, <expr2>, ..., <exprN>
```

In a multiple assignment, all expressions on the right-hand side are evaluated before assigning values to the variables on the left side, and the assignments are considered to happen simultaneously.

Recall from Chapter 6 that a function may return more than one value by separating the values with commas, as on line 18 of Listing 16.1. Each call to `gettag()` in the `process()` function of Listing 16.1 shows how multiple assignment can be used to store multiple return values. Technically, multiple values are returned in a tuple, defined later in Chapter 19.

Constants

Occasionally, it is helpful to create a variable for a value that will never change. Such variables are called **constants**, and in Python they are usually written in all capitals with underscores between words. Listing 16.1 defines the constant `URL` in line 5.

Modules may include names that represent constants in addition to functions. For example, the `math` module includes the constant `pi`. These module constants do not use the all-caps naming convention.

The `string` module provides several constants, including:

`punctuation`	String of all punctuation characters.

Import without From

A second form of **`import`** is used in Listing 16.1:

```
import <module>
```

After this statement, any name from the module may be referred to with dot notation:

```
<module>.<name>
```

One advantage of this form is that all module names are made available without having to list them. A second is that every use of a module name is easy to find because of the dot notation; for example, see line 8. Finally, this syntax prevents accidentally hiding the same name either as a built-in function or from another module. For these reasons, production code usually uses this form, and we will use it frequently throughout the remainder of the text.

Accessing Web Pages

The `urllib.request` module provides support for reading web pages. Every page on the web is described by its **URL** or **uniform resource locator**, essentially its address on the web. Listing 16.1 uses the following function from `urllib.request`:

`urllib.request.urlopen(url)`	File-like object accessing `url`.

The file-like object that is returned supports a `read()` method, but this `read()` method returns raw bytes rather than text. The `str()` type conversion in line 9 converts the bytes to a string.

Exercises

16.1 Use Listing 16.1 to:

(a) Identify the accumulation loop in Listing 16.1, along with the type of accumulator. Explain how you know the type.

(b) Explain why no **`else`** is necessary for the **`if`** in line 15.

(c) Rewrite the **`import`** and **`with`** statements to use **`import from`**.

16.2 Use Listing 16.1 to answer these questions:

(a) Does the program work correctly?

(b) Explain why the previous question is meaningful in a way that is different from all previous code examples.

(c) Determine whether or not the `.strip()` in line 27 is necessary. Explain your conclusion.

16.3 Use Listing 16.1 to answer these questions:

(a) Explain why it is helpful for `gettag()` to return two values instead of one.

(b) Explain how the slice works in line 18 of `gettag()`.

16.4 Although lowercase is preferable, HTML tags may be uppercase, such as `<H2>`. Modify Listing 16.1 to handle tags of either case.

16.5 Modify Listing 16.1 to highlight foods or days you are interested in.

16.6 Modify Listing 16.1 to display information from your own campus or town.

16.7 Modify Listing 16.1 to extract information from a web page of your choice.

16.8 Write a program `weather.py` to print the current weather conditions from the National Weather Service. Ask the user for the zip code, and append their response to the URL `http://forecast.weather.gov/zipcity.php?inputstring=`.

16.9 Write a function `removepunc(s)` that returns a copy of the string `s` with all punctuation removed.

16.10 Look up the string `.translate()` and `.maketrans()` methods and use them to rewrite the previous exercise.

16.11 Write a function `alphaonly(s)` that returns a copy of `s` that retains only alphabetic characters.

16.12 Write the function `mycapitalize(s)` to return a copy of `s` that is capitalized, without using the string `.capitalize()` method. Test your function by comparing it with the built-in method.

16.13 Write the function `mytitle(s)` to return a copy of `s` with each word capitalized, without using the string `.title()` method. Test your function by comparing it with the built-in method.

16.14 Rewrite the `is_dna()` function from Exercise 12.4 to handle both upper and lower case.

16.15 Rewrite the `is_rna()` function from Exercise 12.6 to handle both upper and lower case.

16.16 Rewrite the `transcription()` function from Exercise 12.7 using string methods.

16.17 Rewrite the `countbases()` function from Exercise 12.9 using string methods.

16.18 Look up the string `.translate()` and `.maketrans()` methods and use them to rewrite the `complement()` function from Listing 12.1. Use a constant for the string of nucleotides `"ACGT"`.

16.19 Write a function `acronym(phrase)` that returns the acronym for `phrase`. For example, if the phrase is "random access memory," then the acronym is "RAM;" if the phrase is "as soon as possible," the acronym is "ASAP." Write a `main()` to test your code.

16.20 Write a function `isidentifier(s)` that returns `True` if `s` is a legal Python identifier; otherwise, it returns `False`. Look up the `keyword` module in the documentation and use it to exclude keywords. Write a `main()` to test your function.

16.21 Write a function `ispalindrome(s)` that returns `True` if `s` is a **palindrome**; i.e., a phrase that reads the same backward as forward, excluding any punctuation or whitespace. Write a `main()` to test your code.

16.22 Write a program that finds the longest palindrome in a dictionary.

16.23 Listing 15.1 assumes that the dictionary file and word are all in the same case (upper or lower). Modify the program to remove this assumption and work for dictionaries and words that use any case.

16.24 Modify Exercise 15.3 to use string methods in order to get correct output.

16.25 Modify Exercise 15.9 to ignore punctuation in its calculation.

16.26 Write a program `caesar.py` that encodes and decodes text using a **Caesar cipher**. A Caesar cipher shifts each letter in the message by a fixed number of steps. For example, with a shift of $n = 2$ steps, every "a" becomes a "c," every "b" becomes a "d," and so on, with "z" wrapping around to "b."

Write `encode(msg, n)` and `decode(msg, n)` functions, and test your program by decoding encoded messages and checking that the results are the same as the originals. Maintain case within messages (so that uppercase stays upper and lowercase stays lower), and handle punctuation appropriately.

16.27 Write a program `piglatinfile.py` that asks the user for the name of a file and then translates the entire file into Pig Latin. For extra credit, handle punctuation appropriately. Try your program on books from Project Gutenberg (`http://www.gutenberg.org/`). You may need to run this program from the command line for large files (see page 110).

16.28 Write a program `spellcheck.py` that asks the user for the name of a file and then looks up every word in that file to see if it is in a given dictionary. Print out the words that are not in the dictionary. Handle punctuation appropriately. Warning: this program will take a long time with large files.

This page intentionally left blank

Project: File Compression

Even though storage devices continue to become larger, file compression remains an important tool for everyday computer use. For example, images are often compressed to enable faster transmission across a network or storage on a mobile device.

A simple compression algorithm (see, e.g., [3]) for text files is based on encoding long strings of repeated characters. The idea is to replace a string of repeated characters:

```
!!!!!!!!
```

with an escape sequence made up of an otherwise rare character (such as ~), the count, and then the repeated character:

```
~8!
```

This saves space as soon as there are at least four in a row of any character. Certainly, this type of compression will only benefit some files, but many documents contain, for example, long strings of spaces.

The only difficulty with decompressing a file that has been compressed this way is if the original file contains one of the escape characters (i.e., the tilde ~), because then the decompression program will have difficulty deciding whether to print the escape character or a sequence. One solution is to have the compression program replace any tilde ~ with two tildes ~~; that way, when the decompressor sees a tilde, it can check the next character to see if it is a tilde or a number.

Exercises

1. Describe what happens if this compression algorithm is applied to a file that contains only escape characters (tildes).

2. Write a `compress(text)` function that implements this type of file compression. Use constants for the escape character and minimum number of repetitions. Test your program on itself.

3. Write a `decompress(text)` function that returns compressed text to its original state. You may find it helpful to write a separate function like this:

```
def int_at_start(s):
    # return positive integer at beginning of string s
    #  and index of next character
```

to use after finding an escape character, since the count might be longer than one digit. The index of the next character is used both to find the character to repeat, and to know where to continue processing after the escape.

Be sure that `decompress(compress(text))` is the same as `text`. Test your program on itself.

Chapter 17

Mutable and Immutable Objects

Over the last few chapters, you may have noticed some strange or confusing things about strings and lists. The two types are very similar, but there are significant differences that first appeared when we listed their methods.

Look again at the list of operations in the section "Lists Are Not Like Strings" and at the note below the methods that modify lists on page 78:

> *These methods all modify the list they are called on, and only* `.pop()` *returns anything.*

Compare that with the note below the methods that seem like they should modify strings on page 95:

> *None of these methods changes the original string; they modify and return a copy.*

The list methods change the list and usually do not return anything; the string methods do not change the string but do return a modified copy. It is time to clarify this behavior.

Object State

An object's **state** is the information it knows about itself. For example, a string knows its length and the character in each position. A list knows its length and the object at each position, and an integer knows its value.

An object that may change its state is called **mutable**, while objects that can never change their state are **immutable**. At this point, we have only seen one mutable type:

Mutable	Immutable
List	String
	Integer
	Float
	Boolean

File objects might be considered immutable if opened for reading and mutable if opened for writing, but that terminology is usually not used.

Thus, because strings are immutable, they *never* change. That means we have to work a little harder if we want a string to change. The fact that integers, floats, and boolean values are immutable is not something you need to worry about, because you are already used to assigning a new value if you want a variable of that type to change.

Using Reassignment

Probably the most important technique to remember when trying modify a variable that refers to an immutable object is to **reassign** that variable to the new value:

```
<variable> = <new modified value of variable>
```

For example, the `.upper()` string method only modifies a copy of the string, but if we reassign it like this:

```
s = s.upper()
```

then `s` has essentially been converted to uppercase.

Converting to a Mutable Type

Look again at the `signature()` method of Listing 15.1. The parameter `word` is a string, and so it is immutable. But we wanted to sort its letters, and so we converted the string to a list, which is mutable, and then sorted the list.

Converting the list back to a string is trickier and requires a string method that is the opposite of `.split()`:

`s.join(listofstrs)`	Concatenate `listofstrs` with `s` between each string.

For example,

```
"XYZ".join(["abc", "def", "ghi"]) = "abcXYZdefXYZghi"
```

When `s` is the empty string, `.join()` concatenates all the strings in the list.

Exercises

17.1 Show the output of each of these fragments of Python code. If there is an error, explain its cause.

(a)
```
s = "string methods with strings"
s.replace("string", "file")
print(s.upper())
```

(b)
```
s = "Python is named after Monty."
i = s.find("Monty")
s = s[i:-1] + " " + s[:6].lower()
print(s.title())
```

(c)
```
s = "Python is named after Monty."
if s.lower().startswith("python"):
    s[:6] = "hall"
print(s.capitalize())
```

17.2 Write one line of code to change the third character of a string `s` to a hyphen.

17.3 Modify the `signature()` function in Listing 15.1 to use `.join()` to return a string rather than a list of characters. Does the rest of the program (in particular, `matches()`) still work? Explain why the program does or doesn't work with this change.

17.4 Use the previous exercise to modify Listing 15.1 to find all words in the dictionary file that are their own signatures. Describe the types of words that are found, without using the idea of a signature.

17.5 Write a function `anagram(word1, word2)` that returns `True` if `word1` is an anagram of `word2`; otherwise, it returns `False`.

17.6 Write a function `jumble(word)` that returns a scrambled version of the word (as a string) by randomly permuting its letters.

17.7 Use the `jumble()` function from the previous exercise to write a program `jumbleguess.py` that chooses a secret word from a dictionary, prints a scrambled version of the word, and then asks the user to guess the secret word. Count the number of guesses used.

17.8 Write a function `mutate(dna)` that returns a copy of the string `dna` with one of its nucleotide bases (randomly chosen) randomly mutated to a different base.

17.9 Use the `mutate()` function from the previous exercise to simulate many repeated mutations on a long string of DNA that begins with only one base (such as all G's). How long does it take before the result seems to be like a random string of DNA?

This page intentionally left blank

Project: Hangman

These exercises lead up to developing a Python version of hangman (without the drawings), in which the object is to guess a word by guessing the letters that appear in it.

Software Development

This problem is complex enough that it is difficult to just sit down and write out a complete program. Large, complex projects all use some type of **software development process** to help programmers organize and plan their work. Software design courses study these methodologies and contrast their relative strengths and weaknesses.

Spiral development is software development process that encourages early prototyping and frequent consultation with clients. A **prototype** is an early version of a program that allows programmers and users both to see what the final program might be like. None of our programs will be large enough to require a formal process, but we can benefit from this idea:

1. Begin with a working prototype.

2. Gradually add one feature at a time until the program is complete.

The key is to always have a working program. That makes it much easier to find bugs and finish with a working program. The exercises suggest one way to develop this program in stages.

Exercises

1. Write a program `hangman.py` that chooses a random secret word from a dictionary and allows the user to repeatedly guess a letter in the secret word until they have made 6 guesses. Before each guess, print the number of guesses remaining and the letters guessed so far. At the end, print the secret word. (This game is not very fun.)

2. Modify your program to check to see if the guessed letter is in the secret word. If it is, do not subtract from the number of guesses remaining. In both cases, print an appropriate message.

3. Write a function `template(secret, guesses)` that returns the current state of what would normally be written out in a game of hangman. For example, if the word is "cheese," and the user has guessed "e," "r," and "t," then the template is `"--ee-e"`. Use a string accumulator.

4. Use the `template()` function to finish the game.

5. Modify your program to allow the user to have up to 6 *different* incorrect guesses. If the user guesses a letter that has already been guessed, print a message and allow another guess.

6. Modify your program to display the letters guessed so far in alphabetical order.

Chapter 18

Dictionaries

Imagine writing a program that counts the number of times each word appears in a large file. We could store the counts in a list, but then how do we keep track of which word uses which index? We could store the words in a separate second list, so that the count for the word in `words[i]` is in `counts[i]`, but then we would have to search the entire word list every time we came to another word to count. A dictionary is much more efficient.

Listing 18.1: Word Frequency

```
# wordfreq.py

import string

def getwords(fname):
    with open(fname) as f:
        s = f.read().lower()
    s = s.translate(s.maketrans("", "", string.punctuation))
    return s.split()

def frequency(words):
    d = {}
    for word in words:
        if word in d:
            d[word] += 1
        else:
            d[word] = 1
    return d

def display(d):
    for key in sorted(d):
        print(key, d[key])

def main():
    words = getwords("moby.txt")
    display(frequency(words))

main()
```

The `display()` function in Listing 18.1 is an exception to our habit of printing only in `main()`. However, its name alerts readers that it will produce output.

Command Line Execution

This program may run more slowly inside IDLE than at the **command line**. To run a Python program from the command line in Windows, open a command prompt, use `cd` to navigate to the location of `wordfreq.py`, and type this, changing "`32`" to whatever version you have installed:

```
c:\python32\python wordfreq.py
```

In Linux or OS X, open a Terminal, use `cd` to locate `wordfreq.py`, and run

```
python wordfreq.py
```

Depending on your installation, you may need to use `python3` instead of `python`.

Dictionaries

A **dictionary** is a mapping that stores keys with associated values. For every **key** in a dictionary, there is exactly one **value**. Thus, we think of dictionaries as sets of "key:value" pairs. To look up data in the dictionary, you provide the key, and the dictionary will provide the value.

Dictionaries are written inside curly brackets `{}`, with a colon between each key-value pair. For example, in this dictionary:

```
ages = {"Alice":38, "Bob":39, "Chuck":35, "Dave":34}
```

the keys are the string character names, and each name is associated to one integer value. A pair of empty brackets `{}` denotes an **empty dictionary**.

Dictionaries are also called **associative maps** because they associate keys with values. Key-value pairs are not stored in any particular order in a dictionary, and so dictionaries support different operations and methods than the sequence types we have been using, such as strings, lists, and `range()` objects.

Dictionary keys must be immutable, but there is no requirement that all keys have the same type. Values in a dictionary may be of any type. Dictionaries themselves are mutable objects in Python.

Dictionary Operations

The main operations in a dictionary `d` are to either set or retrieve the value stored for a key:

`d[key] = value`	Set value associated with `key` in `d` to `value`.
`d[key]`	Get value associated with `key` in `d`. Raises `KeyError` exception if `key` not in `d`.

Dictionaries are specifically designed so that the above operations are very fast.

Some of the other operations available are:

`len(d)`	Number of items in `d`.
`del` `d[key]`	Delete entry for `key` in `d`.
`key` **`in`** `d`	`True` if `key` is in `d`.
`key` **`not in`** `d`	`True` if `key` is not in `d`.

Because dictionaries are mutable, there are operations like **`del`** that change the contents of the dictionary.

Dictionary Methods

Dictionaries are objects and therefore also have their own methods:

`d.get(key[, default])`	Get value for `key` if `key` in `d`; otherwise, return `default` (or `None`).
`d.pop(key[, default])`	Get and delete value for `key` if `key` in `d`; otherwise return `default`
`d.keys()`	View of keys in `d`.
`d.values()`	View of values in `d`.
`d.items()`	View of `(key, value)` pairs in `d`.

The `.get()` method is similar to using `d[key]`, except that it returns a default value (or `None`) instead of raising a `KeyError` exception. The `.keys()`, `.values()`, and `.items()` methods return special dictionary **views** that are iterable and dynamic, meaning that they update as the contents of the dictionary changes. The `list()` type converter will convert a view to a list.

Dictionary Loops

Python conveniently allows dictionaries to be used in **`for`** loops:

```
for <variable> in <dict>:    # loop over each key
    <body>
```

Remember that a dictionary's key-value pairs are not stored in any particular order, so this type of loop cannot control the order in which the keys are found.

If you need to loop over a dictionary in sorted order, as in Listing 18.1, use the built-in `sorted()` function:

`sorted(x)`	Sorted list from iterable `x`

Exercises

18.1 Which of the following types may be used as keys in a Python dictionary: integer, float, string, list, boolean, dictionary? Explain your answers.

18.2 Which of the following types may be used as values in a Python dictionary: integer, float, string, list, boolean, dictionary? Explain your answers.

18.3 Write one Python statement to create a dictionary named `filmratings`, where each key is a movie title and the value is your rating of that film from 1 (low) to 5 (high). Include at least four entries.

18.4 Use Listing 18.1 to answer these questions:

(a) Describe the purpose of line 8. Look up the `.translate()` and `.maketrans()` string methods if you have not used them before.

(b) Explain why the **`if`** statement is necessary at line 14.

(c) Describe the effect of removing the call to `sorted()` in line 21.

(d) Give the type of data referred to by `words` in line 26. Explain how you know.

18.5 As it is, Listing 18.1 does not have a separate function to remove punctuation, even though that work is being done. Rewrite the program to use a separate `removepunc(s)` function.

18.6 Rewrite the `frequency()` function in Listing 18.1 to remove the **`if`** statement by using `.get()` with a default value. Discuss the tradeoffs.

18.7 Research the `defaultdict()` function from the `collections` module and use it to rewrite the `frequency()` function in Listing 18.1. Discuss the tradeoffs.

18.8 Listing 18.1 makes some poor decisions while counting word frequencies. Fix how each of these is handled:

(a) Hyphenated words

(b) Apostrophes

(c) Dashes (`"--"` in Moby Dick)

Doing Exercise 18.5 first may be helpful.

18.9 Write a program `letterfreq.py` that computes the frequency with which each letter appears in a given file. Handle case appropriately. Letter frequencies are important for encryption, compression, modern keyboard designs, and some games.

This exercise provides an alternate solution to Exercise 12.9 to count the nucleotide bases in a DNA string.

18.10 Write a program `dict.py` that provides an interactive dictionary, where a user can add or change definitions, as well as look up definitions. The program starts with an empty dictionary, and then entries are gradually created by the user. An interactive session might look like this:

```
Welcome to PyDict
        [a]dd or change a definition
        [l]ookup a word
        [q]uit
Your choice: l
Word to look up: python
Not in dictionary
        [a]dd or change a definition
        [l]ookup a word
        [q]uit
Your choice: a
Word: python
Definition: A long, large snake.
        [a]dd or change a definition
        [l]ookup a word
        [q]uit
Your choice: l
Word to look up: python
Definition: A long, large snake.
        [a]dd or change a definition
        [l]ookup a word
        [q]uit
Your choice: q
Thanks!
```

18.11 Modify Exercise 15.12 to use a dictionary in order to find the word(s) with the most anagrams. Does this approach result in a faster program?

This page intentionally left blank

Project: ELIZA

In the mid-1960s, Joseph Weizenbaum at MIT wrote ELIZA, a groundbreaking program in **artificial intelligence** that allowed users to converse with it using plain English. ELIZA was meant to sound like a Rogerian therapist: friendly, non-judgmental, and reflective, using questions to get the user to talk further about him or herself. By searching for certain keywords, ELIZA could provide very focused responses, in addition to using a variety of generic statements and questions.

The web has many examples of similar programs now, also known as chatterbots.

Exercises

1. Research the Turing Test. Report your findings, including its connection with ELIZA.

2. Write a program that implements your own idea for an ELIZA. In other words, try to develop a specific personality, in the way that Weizenbaum tried to emulate a therapist.

   Use spiral development (see page 107). Begin with a very simple working program and gradually add features like these:

   - Find out the user's name and use it occasionally.
   - Use keywords to provide appropriate responses.
   - Have a supply of random responses to fall back on if none of the keywords match.
   - Look for likely types of responses or questions.

This page intentionally left blank

Project: Shannon Entropy

In 1948, Claude Shannon founded the field of **information theory** with his paper, "A Mathematical Theory of Communication" [5]. In it, he defined the **entropy** H of a message as

$$H = -\sum_i p_i \log_2(p_i)$$

where p_i is the probability of the ith character occurring in the message. This **probability** can be easily calculated if we count the number of times each character appears in the message:

$$p_i = \frac{\text{number of times } i\text{th character appears}}{\text{length of message}}$$

When entropy H is calculated as above, using the log base two, it measures the average number of bits[1] per character required to communicate the given message.

Example: Low Entropy

Intuitively, in a string of 26 e's alternated with 26 f's, neither character carries very much information, because they act so similarly. To calculate the entropy of this string, we first calculate the probabilities of each character. The probability of an "e," p_e, is:

$$p_e = \frac{\text{count of e's}}{\text{length of message}} = \frac{26}{52} = 0.5$$

and the probability of an "f" is exactly the same. Thus, the entropy is:

$$H = -0.5 \log_2(0.5) - 0.5 \log_2(0.5) = 1$$

which says that each character of this message carries an average of one bit's worth of information.

[1] Interesting fact: Shannon's paper introduced the term "bit."

Example: Higher Entropy

If we now calculate the entropy of the string "abcdefghijklmnopqrstuvwxyz-abcdefghijklmnopqrstuvwxyz," each character has probability:

$$p = \frac{2}{52} = \frac{1}{26}$$

and the entropy is:

$$H = -26 \left(\frac{1}{26} \log_2 \frac{1}{26} \right) \doteq 4.7$$

which is much higher than the previous example. In this case, each character requires an average of at least 4.7 bits to encode it.

Shannon entropy gives a limit to how much compression is possible with a file: if each character requires, as in the last example, an average of 4.7 bits, then one could not compress that text to fewer than approximately $4.7 \times 52 = 244.4$ bits.

Logs

The `log()` function from the `math` module allows the base to be specified as an optional second argument:

`log(x[, b])`	Log of `x` base `b`. Default base is e.

Exercise

1. Write a program `entropy.py` to calculate the Shannon entropy of a text file. Compare the required average number of bits per character to the number of bits per character actually used if the file is stored as ASCII.

Project: Reading DNA Frames

Exercise 12.7 describes (without the chemistry) how DNA is converted to RNA through transcription. After transcription, RNA is converted to a sequence of amino acids through a process called **translation**. RNA translation takes groups of three nucleotide bases called **codons**, and converts each codon to an amino acid according to the following table (see, e.g., [4]):

	Base 2				
Base 1	**U**	**C**	**A**	**G**	**Base 3**
U	F	S	Y	C	U
	F	S	Y	C	C
	L	S	_	_	A
	L	S	_	W	G
C	L	P	H	R	U
	L	P	H	R	C
	L	P	Q	R	A
	L	P	Q	R	G
A	I	T	N	S	U
	I	T	N	S	C
	I	T	K	R	A
	M	T	K	R	G
G	V	A	D	G	U
	V	A	D	G	C
	V	A	E	G	A
	V	A	E	G	G

To look up, say, the translation of AGC, find A in the first column, then look down the G column (as the second base), and find the C row by using the last column. This gives the amino acid S, which is serine. The table uses standard one-letter abbreviations for the 20 amino acids that you can look up if you are interested. A few codons signal a `STOP` in the sequence; these are indicated by _'s in the table.

Reading Frames

Proteins are encoded in this way as strings of amino acids. Given a strand of RNA, there is no way to know ahead of time where to start reading the

3-base codons. Thus, we need to try all three possibilities to find possible proteins: starting at the first base (index 0), second base (index 1), or third (index 2). Each of these is called a **reading frame**. **Open reading frames** are reading frames that have long stretches without `STOP` codons, and these are good candidates for protein-coding sequences.

Given a strand of DNA, as opposed to RNA, remember that there are two strands bound together in a double-helix: the given DNA strand and its reverse complement. Thus, to search a segment of DNA for protein sequences requires checking *six* reading frames: three based on the given strand, and three from its reverse complement.

FASTA Data Files

FASTA is an easy-to-read data file format for DNA sequences. The first line of the file is a **header line** that begins with a `>` and can be ignored. The remaining lines of the file contain the DNA sequence.

Exercises

1. Write a `translate(rna, frame)` function that translates one frame (0, 1, or 2) of RNA to the corresponding sequence of amino acids.

2. Write a `readfasta(fname)` function that returns the string of DNA from a FASTA file.

3. Develop a program `amino.py` to read a DNA sequence from a FASTA file and print all six of its reading frames. Convert `STOP` codons to newlines so that proteins are easier to find.

Part III

Selected Topics

This page intentionally left blank

Chapter 19

Sound Manipulation

Sounds are represented in computer memory as a sequence of discrete samples of air pressure readings. CDs are recorded using 44,100 of these samples per second. With that many samples, it is possible to reconstruct the original sound waves in such a way that our ears have a difficult time hearing any difference between the reproduction and the original sound. Given access to those samples in a Python program, we can do some interesting things.

Listing 19.1: Reverse WAV

```
# reversewav.py
import array
import contextlib
import wave

def datatype(width):
    return "B" if width == 1 else "h"

def readwav(fname):
    with contextlib.closing(wave.open(fname)) as f:
        params = f.getparams()
        frames = f.readframes(params[3])
    return array.array(datatype(params[1]), frames), params

def writewav(fname, data, params):
    with contextlib.closing(wave.open(fname, "w")) as f:
        f.setparams(params)
        f.writeframes(data.tostring())
    print(fname, "written.")

def main():
    fname = input("Enter the name of a .wav file: ")
    data, params = readwav(fname)
    outfname = "rev" + fname
    writewav(outfname, data[::-1], params)

main()
```

Before running this program, put a WAV file with name ending in `.wav` in the same folder or directory as the program, and provide that file name when the program requests it. Then listen to the file that was created. See page 71 if you do not recognize the conditional expression on line 7.

WAV Files

The `wave` module provides one key function:

`wave.open(file[, mode])`	Open `file` in mode `"r"` (read, the default) or `"w"` (write).

The `wave.open()` function returns either a `Wave_read` or `Wave_write` object. If `f` is a `Wave_read` object, then it has these methods:

`f.getparams()`	Get tuple of parameters describing `f`.
`f.readframes(n)`	Read `n` frames from `f`.

The parameters returned by `.getparams()` are described below. There is also a separate `.get...()` method for each of the individual parameters.

If `f` is a `Wave_write` object, then it has these corresponding methods:

`f.setparams(params)`	Set parameters for `f` to `params`.
`f.writeframes(frames)`	Write frames to `f`.

The argument to `.setparams()` may be a tuple or list. There is a separate `.set...()` method for each of the individual parameters. In Listing 19.1, the output file parameters are set to be the same as those that were used in the input file.

WAV File Parameters and Frames

Understanding the parameters that control WAV files and their relation to the frames returned is the key to using them correctly. A call to `.getparams()` returns these six values:

```
(nchannels, sampwidth, framerate, nframes, comptype, compname)
```

nchannels: The number of **channels** per frame: 1 for mono recordings, 2 for stereo. If there are 2, the samples alternate left, right, left, right, etc.

sampwidth: The number of bytes per sample. One-byte samples are unsigned integers 0 to 255, whereas two-byte samples are signed integers $-32,768$ to 32,767.

framerate: Also known as the **sampling frequency**, this is the number of frames per second (44,100 for CD quality).

nframes: The number of frames in the file.

For example, if `nchannels = 2` and `sampwidth = 2`, then each frame is 4 bytes long and made up of 2 samples, left followed by right. If `nchannels = 1` and `sampwidth = 1`, then each frame is a single sample made up of one byte.

The last two parameters refer to compression; we will not use them.

Tuples Are Immutable Lists

Both `.getparams()` and `.setparams()` use a tuple to communicate all six parameters at once. In Python, a **tuple** is exactly like a list except that it is immutable.

Tuples are written with parentheses instead of square brackets: for example, `t = (1, 2, 3)` is a tuple, whereas `u = [1, 2, 3]` is a list. Individual entries may be accessed by index, as in lines 12 and 13 of Listing 19.1.

It is sometimes convenient to **unpack** a tuple into its separate entries using multiple assignment:

```
<var1>, <var2>, ..., <varN> = <tuple>
```

Each variable on the left is assigned to the corresponding value from the tuple on the right, as long as the number of items match.

Recall our earlier discussion of multiple return values and multiple assignment (see page 95). In the terminology of this chapter, functions return more than one value by returning a tuple, and that tuple is often unpacked by the caller.

Data Arrays

The `array` module converts the raw bytes in a sound sample into an array that is easy to manipulate.

`array.array(code, rawdata)`	Array initialized with `rawdata` according to `code`.

The type codes corresponding to data in WAV files are:

`B`	Unsigned 1-byte integer.
`h`	Signed 2-byte integer.

The array object returned by `array.array()` supports indexing, slicing, concatenation, repeated concatenation, and most of the same methods as lists. In other words, once you have sound data in an array, you can manipulate it just as you would a list. If `data` is an array, you can find out the type code it was created with like this:

`data.typecode`	Type code used to create `data`.

This is technically a field of the array object, which you will learn about in Chapters 21 and 23.

Context Managers

Finally, Listing 19.1 requires the `contextlib` module, which provides methods that objects need in order to be used inside **`with`** statements. In this case, we need a closing function because without it, `Wave_read` and `Wave_write` objects do not know to call their `.close()` method when the **`with`** block completes.

`contextlib.closing(obj)`	Manager to close `obj` when block finishes.

Exercises

19.1 Use Listing 19.1 to:

(a) Identify where in the code the sound samples are reversed.

(b) Explain the purpose of the `datatype()` function.

(c) Identify the WAV parameter returned by `params[3]` in line 12.

(d) Identify the WAV parameter returned by `params[1]` in line 13.

19.2 Suppose `params` refers to a tuple of WAV file parameters, as in line 11 of Listing 19.1. Write one line of code to unpack `params` into its six components.

19.3 Write one line of code to pack the variables `nchannels`, `sampwidth`, `framerate`, `nframes`, `comptype`, and `compname` into a tuple named `params`.

19.4 Explain why two sets of parentheses are necessary in this method call:

```
f.setparams((nchannels, sampwidth, framerate,
             nframes, comptype, compname))
```

What could the inner set of parentheses be replaced with?

19.5 Modify Listing 19.1 to use a separate `reverse(data)` function.

19.6 Write a function `display(params)` that prints the parameter list in a nice format. Include the duration of the WAV file in seconds.

19.7 Write a function `silence(typecode, length)` that returns a new data array containing all zeros of the given type code and length.

19.8 Write a function `left(data)` that returns the left channel of a two-channel stereo data array.

19.9 Write a function `right(data)` that returns the right channel of a two-channel stereo data array.

19.10 Write a function `mix(left, right)` that mixes left and right channel data arrays into one two-channel stereo data array. Do not assume the two channels have the same length.

19.11 Listing 19.1 does not handle stereo sounds correctly. Describe what it currently does and rewrite the program to work for both mono and stereo files. Exercises 19.7–19.10 will help.

19.12 Write a function `clamp(x, a, b)` that returns the value of `x` **clamped** to be in the interval $[a, b]$:

$$\text{clamp}(x, a, b) = \begin{cases} a & \text{if } x <= a \\ b & \text{if } x >= b \\ x & \text{otherwise} \end{cases}$$

Hint: it can be done in one line using `min()` and `max()`.

19.13 Write a function `clampsample(value, typecode)` that uses the previous exercise to return `value` clamped into the appropriate range for type codes "B" and "h." Because samples must be integers while `value` may be a float, have this function return an integer.

19.14 Write an `amplify(data, factor)` function that returns a copy of `data` with each sample multiplied by `factor`. Do not change the original `data` array, and use clamping to make sure each new sample has a valid value.

19.15 Write a function `maxabs(data)` that returns the maximum of the absolute value of the elements in `data`. Note that this is just the `max()` if `data` has type code "B."

19.16 Write a function `normalize(data)` that uses amplification to normalize `data`. A sound file is **normalized** if it uses the full range of available values for its sample width. Use `maxabs()` from the previous exercise to help calculate the amplification factor.

19.17 Write a function `addsamples(data1, data2)` that creates a new data array containing the sum of the samples in the given arrays. Assume `data1` and `data2` have the same type code, but do not assume they are the same length. Clamp the new values.

19.18 Write a function `echo(data, delay, factor)` that returns a copy of the original sound with an echo. The `delay` parameter specifies the number of frames that the echo is delayed, and `factor` is the amplification factor for the echo (less than 1).

The data array returned by this function will be longer than the original because of the delayed echo, so you will not be able to use the original file's `params` when you write the new file. You will need to send a new set of parameters (which may be in a list) to write the new file.

This page intentionally left blank

Chapter 20

Sound Synthesis

Now that we know how samples are used to represent sounds in a WAV file, we can write programs to create our own sounds. In other words, we can create **synthesizers**.

This program listing is not complete: you will need to add appropriate imports, the `writewav()` function from Listing 19.1, and `clampsample()` from Exercise 19.13.

Listing 20.1: Synthesizer

```
# synth.py

# imports and some function definitions deleted

def sinenote(step, sec, sampfreq):
    data = array.array("h")
    samples = int(sampfreq * sec)
    freq = 440 * 2 ** (step / 12)
    for i in range(samples):
        y = 32767 * sin(2.0 * pi * freq * i / sampfreq)
        data.append(clampsample(y, "h"))
    return data

def notes(notefn, steps, sec, sampfreq):
    data = array.array("h")
    for step in steps:
        data += notefn(step, sec, sampfreq)
    return data

def main():
    sampfreq = 11025
    pisteps = [3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5]
    data = notes(sinenote, pisteps, 0.4, sampfreq)
    params = [1, 2, sampfreq, len(data), "NONE", None]
    writewav("synth.wav", data , params)

main()
```

Sound Waves

Synthesizers often use mathematical functions like the sine wave as a basis for generating sounds. In fact, a sine wave is considered a "pure" tone because it has a single frequency with no additional harmonics. You can hear this purity in the tones generated by Listing 20.1. The main work involved in creating these sounds is in getting the right samples of the right sine waves.

Musical Notes and Frequencies

Listing 20.1 bases its synthesis on the A above middle C, known as **A440** because its frequency is generally 440 Hz. The unit **Hz** (hertz) measures cycles per second, so the pitch A440 is generated by any wave that repeats at 440 cycles per second. Doubling the frequency of a note produces a tone one octave higher than the original; halving it moves one octave lower. Thus, 880 Hz is the frequency of an A one octave higher than A440.

The relationship between the frequencies of adjacent notes in the chromatic scale is more complicated than it is for octaves. To move up a half-step (to the next note on a keyboard) from a frequency f,

$$1 \text{ half-step up from } f = (2^{\frac{1}{12}}) \times f$$

Moving 5 half-steps up means repeating this multiplication 5 times:

$$\begin{aligned} k \text{ half-steps up from } f &= (2^{\frac{1}{12}})(2^{\frac{1}{12}})(2^{\frac{1}{12}})(2^{\frac{1}{12}})(2^{\frac{1}{12}}) \times f \\ &= (2^{\frac{5}{12}}) \times f \end{aligned}$$

There are twelve half-steps in an octave, so the calculation $(2^{\frac{12}{12}})f = (2^1)f = 2f$ is consistent with the frequency doubling when moving up one octave.

Thus, the `step` parameter of the `sinnote()` function specifies the number of steps above or below A440 for the note to be generated.

Frequency and Sine Functions

Recall that the period and frequency of any wave are inversely related:

$$\text{frequency} = \frac{1}{\text{period}}$$

Therefore, to find a sine wave with frequency 440 Hz, we need a wave with period 1/440 of a second. We can find such a wave by gradually working up to it:

Function	Period	Frequency
$\sin(x)$	2π	$1/2\pi$
$\sin(2\pi x)$	1	1
$\sin(2\pi \cdot 440 \cdot x)$	1/440	440

Thus, for any f,

$$y = \sin(2\pi \cdot f \cdot x)$$

has frequency f.

Exercises

20.1 Use Listing 20.1 to answer these questions:

(a) Explain the purpose of the `notes()` function.

(b) Explain the purpose of the `sinenote()` function.

(c) Explain each term in the calculation of the number of samples in line 7.

(d) Explain the 32767 in line 10.

(e) Explain the `i / sampfreq` term in line 10.

(f) Explain the meaning of the values of `params[0]` and `params[1]` in line 24.

20.2 Use Listing 20.1 to answer these questions:

(a) Identify the notes that are played by this program. For example, the first note is a C, one octave above middle C.

(b) Speculate as to what might cause the clicks between notes.

20.3 Modify Listing 20.1 to use a sample width variable so that files with sample width 1 or 2 may be generated. Width 1 samples are always positive, so you will need to shift the sine wave up in that case to get complete cycles. Output files should sound similar for both widths.

20.4 Modify Listing 20.1 so that an amplitude (volume) parameter may be given to the `notes()` and `sinenote()` functions. This parameter should be a multiplicative factor that works like the factor in Exercise 19.14.

20.5 Rewrite the `silence()` function from Exercise 19.7 to have three parameters: seconds, width, and sample frequency. That will make it easier to use with the note generation functions of this chapter.

20.6 Write a `squarewave(x)` function that computes the square wave function, whose graph looks like this:

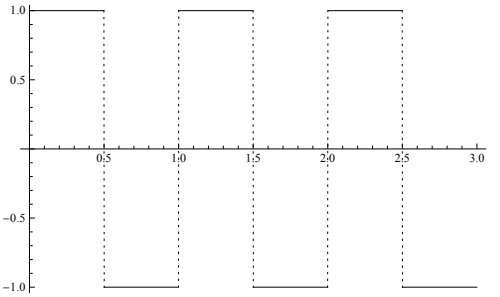


Figure 20.1: Square wave.

Hint: consider the effect of `x % 1` on floating-point values.

20.7 Use the previous exercise to write a `squarenote()` function that generates one note using a square wave instead of a sine wave.

20.8 The previous exercise likely caused you to write a `squarenote()` function that has a lot of the same code as the `sinenote()` function of Listing 20.1. **Refactor** your code to put as much of the common code as possible into a new function named `onenote()` that takes one new parameter: the function (`sin()` or `squarewave()`) to apply.

One difficulty is that these two functions have different periods, so there is an extra 2π in the sine wave generator. Write a `sinwave(x)` function that computes a sine wave of period 1.0. Then the `onenote()` function can assume that its function parameter always has period 1.0.

20.9 Write a `trianglewave(x)` function that computes the triangle wave function, whose graph looks like this:

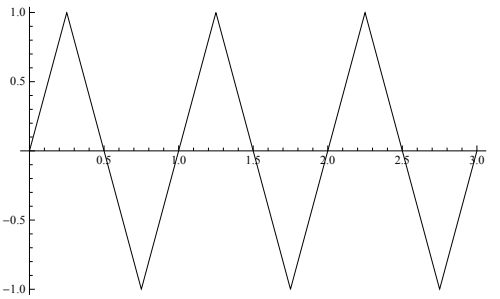


Figure 20.2: Triangle wave.

Use this function to generate sounds synthesized from triangle waves.

20.10 Write a `sawtoothwave(x)` function that computes the sawtooth wave function, whose graph looks like this:

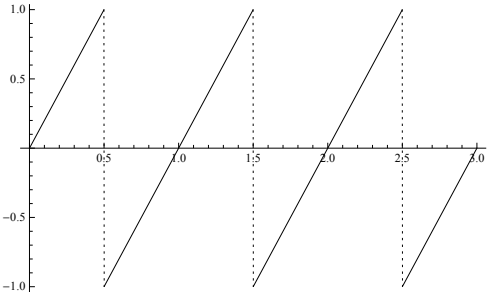


Figure 20.3: Sawtooth wave.

Use this function to generate sounds synthesized from sawtooth waves.

20.11 Write a `whitenoise(x)` function that computes a random value in the interval $[-1, 1]$. Use this function to generate white noise.

20.12 Create an interesting WAV file using the tools from this chapter and Chapter 19.

This page intentionally left blank

Chapter 21

Image Manipulation

Like sounds, digital images are **discretized** versions of their analog counterparts. Regular samples of color (instead of air pressure) are stored in a two-dimensional grid (instead of a one-dimensional array). Access to these samples allows us to change them in a variety of ways.

The following program requires `image.py` from Listing 24.3 to be in the same folder or directory.

Listing 21.1: Two-Tone Image

```
# twotone.py

from image import ImagePPM

def luminance(c):
    r, g, b = c
    return 0.2 * r + 0.7 * g + 0.1 * b

def twotone(c, bright, cutoff, dark):
    return bright if luminance(c) > cutoff else dark

def f(c):
    return twotone(c, (255, 255, 255), 110, (0, 30, 70))

def applyfn(f, img):
    width, height = img.size
    for i in range(width):
        for j in range(height):
            img.putpixel((i, j), f(img.getpixel((i, j))))

def main():
    img = ImagePPM.open("sup.ppm")
    applyfn(f, img)
    img.save("sup2.ppm")

main()
```

Computer Memory: Images

Computer programs represent images as two-dimensional grids of **pixels**. Pixel locations are given by (i, j) coordinates with $(0, 0)$ at the upper left corner and the j coordinates increase as you move down. For example, the pixel coordinates for a tiny 5×5 image are:

$(0,0)$	$(1,0)$	$(2,0)$	$(3,0)$	$(4,0)$
$(0,1)$	$(1,1)$	$(2,1)$	$(3,1)$	$(4,1)$
$(0,2)$	$(1,2)$	$(2,2)$	$(3,2)$	$(4,2)$
$(0,3)$	$(1,3)$	$(2,3)$	$(3,3)$	$(4,3)$
$(0,4)$	$(1,4)$	$(2,4)$	$(3,4)$	$(4,4)$

Notice the differences between these coordinates and mathematical graphs in the xy-plane: the origin is in the upper left corner instead of lower left, and the j coordinate grows down instead of up.

Each pixel in a 24-bit **RGB image** has one byte of storage for each of three components: red (R), green (G), and blue (B). Each byte is interpreted as an unsigned integer, so its value is between 0 and 255. At 3 bytes per pixel, image files become large very quickly, so they are usually compressed.

RGB Color

Python programs represent RGB values as integer tuples. Common RGB colors are:

RGB Tuple	Color
(0, 0, 0)	black
(255, 0, 0)	red
(0, 255, 0)	green
(0, 0, 255)	blue
(255, 255, 0)	yellow
(255, 0, 255)	magenta
(0, 255, 255)	cyan
(255, 255, 255)	white

RGB values represent the intensities of red, green, and blue wavelengths of light, and so $(0, 0, 0)$ is the darkest color because each wavelength has no intensity, and $(255, 255, 255)$ is the brightest because each wavelength is at full intensity.

$\Longrightarrow$ Caution: Mixing light is different than mixing paint. Light colors are additive, whereas paint colors are subtractive.

PPM Images

Many common image formats such as JPEG, GIF, and PNG compress image data in order to save space. Processing these files requires complex compression

and decompression algorithms, which are beyond the scope of this course. **PPM** files contain uncompressed RGB image data which is easy to manipulate in code and can be viewed in a text editor. Thus, Listing 21.1 uses PPM rather than a more complex format. Use a tool such as GIMP, available at `http://www.gimp.org`, to convert your own images to PPM (ASCII) format. See page 167 for more information on the PPM format.

ImagePPM Objects

Listing 21.1 uses a new data type called `ImagePPM` to represent a PPM image. This is not a built-in type; it is defined in a module called `image`, and you can see its code in Listing 24.3. For now, you just need to know how to use `ImagePPM` objects.

There are two ways to create a new `ImagePPM` object, either from an existing PPM file or as a brand new image:

`ImagePPM.open(fname)`	Open an existing image file.
`ImagePPM.new((width, height))`	Create new (all black) PPM image of the given size.

$\Longrightarrow$ Caution: The parameter of `Image.new()` is a tuple, so it looks like there is an extra set of parentheses.

Both `ImagePPM.open()` and `ImagePPM.new()` return an `ImagePPM` object that responds to these methods:

`img.getpixel((i, j))`	Get RGB tuple at `(i, j)`.
`img.putpixel((i, j), (r, g, b))`	Set pixel `(i, j)` to `(r, g, b)`.
`img.save(fname)`	Save `img` in file `fname`.

The `.getpixel()` and `.putpixel()` methods take tuple parameters.

Modifying Images

Listing 21.1 provides a framework for writing many image manipulation programs. The nested loops in `applyfn()` are key:

```
for i in range(width):
    for j in range(height):
        # modify pixel at location i, j
```

They allow us to access every pixel in the image.

Luminance

The physical basis for RGB color is that human eyes have three types of cones that respond to different wavelengths of color: one centered around red, one green, and one blue. These cones have noticeably different sensitivities, so that

our eyes are much more sensitive to green light than red, and more sensitive to red light than blue.

Luminance is an attempt to estimate the brightness of a color as perceived by our eyes. An approximate value based on current display technology (see, e.g., [1]) is

$$\text{luminance}(r, g, b) = 0.2 * r + 0.7 * g + 0.1 * b$$

The coefficients 0.2, 0.7, and 0.1 add up to 1.0, so that the result stays between 0 and 255.

Object Fields

Every object in Python stores its data in variables called **fields** or **instance variables**. Most of the time, we access that data through an object's methods, but occasionally there is a need to access a field directly. The syntax to access an object's field is:

```
<object>.<field>
```

This is identical to the syntax for calling a method except that there are *no parentheses* after the name of a field.

Chapter 19 introduced the `typecode` field of an array; recall that it used this dot syntax with no parentheses. Images have an important field that we need to access when opening an existing image:

`img.size`	Tuple `(width, height)` giving size of `img` in pixels.

Line 16 unpacks the tuple, although we could also have used `img.size[0]` and `img.size[1]`.

What about f?

Finally, the function `f` in Listing 21.1 probably looks a little strange. What is its role?

The related functions `applyfn()` and `twotone()` should be easier to follow. The first allows us to apply any function to each pixel of an image; that should be useful. The second computes a two tone color for an input color `c` based on its luminance. The issue is that the `f()` that `applyfn()` needs can only have *one* parameter, and `twotone()` has four parameters. This isn't really a problem, because any time we want to apply a two-tone, we'll want to specify the other three parameters, anyway: `bright`, `cutoff`, and `dark`. So, that is what `f()` does. It creates a one-parameter function from `twotone()` by specifying values for `bright`, `cutoff`, and `dark`.

Python provides **`lambda`** expressions exactly for situations like this. Exercise 21.4 asks you to try it as an alternative to writing `f()`.

Exercises

21.1 Use Listing 21.1 to:

(a) Describe the order that pixels are visited in the nested loops of lines 17 and 18.

(b) Rewrite the `luminance()` function to not unpack the tuple `c`. Discuss the tradeoffs.

(c) Rewrite the `twotone()` function to use an **`if`** statement instead of a conditional expression. Discuss the tradeoffs.

21.2 Create your own interesting two tone image.

21.3 Write a `threetone(c, bright, cutoff1, med, cutoff2, dark)` three tone function and use it to create an interesting three tone image.

21.4 Research the Python **`lambda`** form and use it to rewrite Listing 21.1 with a lambda expression instead of `f()`. Discuss the tradeoffs.

21.5 Shades of gray have RGB values with all three components the same. **Grayscale** images contain only shades of gray. Implement these two ways of writing a `grayscale(c)` function to create grayscale images from color images:

(a) Compute the gray shade of a pixel using the average of its R, G, and B components.

(b) Compute the gray shade of a pixel using its luminance.

Discuss the results.

21.6 Write a boolean function `isgray(c)` that returns `True` if `c` is a shade of gray (including black or white) and otherwise returns `False`.

21.7 The **negative** of an RGB color is given by $(255 - r, 255 - g, 255 - b)$. Write a `negative(c)` function that returns the negative of the color `c` and use it to create an interesting negative image.

21.8 Write a `brighten(c, s)` function that brightens the color `c` if `s` is positive and darkens it if `s` is negative. Treat `s` as a percentage, so that `brighten(c, 0.3)` brightens `c` by 30 percent. Do not worry about luminance for this exercise. Use your function to brighten or darken an image.

21.9 Write a `brg(c)` function that rotates the color components (r, g, b) to (b, r, g) and use it to create an interesting image.

21.10 Write an `addnoise(c, amt)` function that adds a small amount of randomness (between 0 and `amt`) to each component of the color `c`. Add a different random amount to each component. Use this function to add artificial noise to an image.

> The preceding exercises only modify color based on each pixel's own previous value and so may use `applyfn()` as is. However, the exercises that follow require creating a separate new image using `ImagePPM.new()`. Approach writing these functions by starting with the body of `applyfn()` and then modifying that code to create and return the appropriate new image.
>
> Functions that change the size of the image can also usually be written in two ways: one that loops over the old size and one that loops over the new size.

21.11 Write a function `crop(img, upperleft, lowerright)` that returns a new image consisting of the rectangle in `img` defined by the two points that are given as parameters. Assume `upperleft` and `lowerright` are tuples with two entries each (their x and y coordinates).

21.12 Write a function `double(img)` that returns a copy of `img` twice as wide and twice as tall as the original.

21.13 Write a function `half(img)` that returns a copy of `img` half as wide and half as tall as the original. Scale down by choosing one pixel from each 2×2 block.

21.14 Write a function `scaleup(img, n)` that returns a copy of `img` `n` times as wide and `n` times as tall as the original. Assume `n` is an integer.

21.15 Write a function `scaledown(img, n)` that returns a copy of `img` $1/n$ times as wide and $1/n$ times as tall as the original. Assume `n` is an integer. Scale down by choosing one pixel from each $n \times n$ block.

21.16 Write a function `scaledownavg(img, n)` that returns a copy of `img` $1/n$ times as wide and $1/n$ times as tall as the original. Assume `n` is an integer. Scale down by averaging the colors of all pixels in each $n \times n$ block. Compare the results with choosing one pixel from each block.

21.17 Write a function `pixelate(img, n)` that first scales the image down and then scales the result back up to the original size. The result should be a copy of `img` that is the same size as the original but looks pixelated, with a lower resolution.

21.18 Write a function `tile(img, n)` that returns a new image containing n^2 full-size copies of `img` tiled to make an `n`-by-`n` square.

21.19 Write a function `albumcover(img)` that combines tiling with scaling down to create a tiled copy of `img` that is the same size as the original.

21.20 Write a function `rotate_r(img)` that returns a copy of `img` rotated 90 degrees to the right.

21.21 Write a function `rotate_l(img)` that returns a copy of `img` rotated 90 degrees to the left.

21.22 Write a function `fliptb(img)` that returns a copy of `img` reflected across a horizontal line through the center of the image.

21.23 Write a function `fliplr(img)` that returns a copy of `img` reflected across a vertical line through the center of the image.

21.24 Write a function `mirrortb(img)` that returns a copy of `img` with the top half mirrored onto the bottom half.

21.25 Write a function `mirrorlr(img)` that returns a copy of `img` with the left half mirrored onto the right half.

21.26 Write a function `mirrorbt(img)` that returns a copy of `img` with the bottom half mirrored onto the top half.

21.27 Write a function `mirrorrl(img)` that returns a copy of `img` with the right half mirrored onto the left half.

21.28 Create your own image transformation function and use it to modify an image in an interesting way.

This page intentionally left blank

Project: Image Filters

Many common image manipulations can be described as the application of a filter. An **image filter** is a square array of coefficients that describes how old pixel values will be combined to find a new pixel value. Each entry in the array is a coefficient to be multiplied by the color in that location. The resulting RGB values are then added together to produce the final result. The center of the filter corresponds to the position of the pixel whose color is currently being computed.

For example, this is a 3×3 filter that takes 0 times the current color (in the center) plus 1/8 times each of the colors in the surrounding pixels:

1/8	1/8	1/8
1/8	0	1/8
1/8	1/8	1/8

The fraction 1/8 is used so that the resulting image has approximately the same brightness as the original, because the sum of the pixel values from the filter is one "pixel's worth" of color. The effect of this filter is to blur the original image since the (possibly) distinctive color that used to be in the center location has been replaced by a combination of all the colors around it.

The following filter performs a type of edge detection by highlighting whatever is different in the center pixel while removing the colors that surround it:

0	−1	0
−1	4	−1
0	−1	0

It gives a combined zero pixel's worth of color, which means that the resulting image will be quite dark. Either integers or fractions may be used in filters, and coefficients may be positive, negative, or zero.

Applying Filters

In order to apply a filter to an image, the **for** loops that iterate over all pixels in the original image must be adjusted to avoid the edges. At each pixel, we need to be able to access the pixels above, below, left, and right of it. So, for a 3×3 filter, the main loops look like this:

```
for i in range(1, width-1):
    for j in range(1, height-1):
        # apply filter to pixel (i, j) in original image
```

More refined effects may be achieved with 5×5 filters, and they require moving in two pixels from every edge.

Exercises

1. Write a function `applyfilter(f, img)` that returns a new image created by applying the filter `f` to the given image `img`. Assume `f` returns an RGB tuple and takes three parameters: the original image `img` and the coordinates i and j.

   $\Longrightarrow$ Caution: Do not modify the original image, because the filter depends on having the old values at every pixel.

2. Write a function `createfilter(coeff, pixels, i, j)` that can be used to create filter functions to pass to `applyfilter()`. The `coeff` parameter is a list of the filter coefficients in this order:

0	1	2
3	4	5
6	7	8

   The idea is that by writing this function you can then define a specific filter like this:

   ```
   def blur(img, i, j):
       return createfilter([1/8, 1/8, 1/8,
                            1/8, 0, 1/8,
                            1/8, 1/8, 1/8], img, i, j)
   ```

   Then the `blur()` function could be passed to `applyfilter()`.

   The `createfilter()` function contains all the messy work of the filter calculation. For each component (R, G, B), it computes the sum over the 3×3 grid:

   ```
   coeff[0] * img.getpixel((i - 1, j - 1)) +
       coeff[1] * img.getpixel((i, j - 1)) + ...
   ```

   By putting this calculation in one place, you do not have to rewrite it for every different filter. Remember that image RGB components must be integers.

3. Implement these filters:

   (a) Blur

1/8	1/8	1/8
1/8	0	1/8
1/8	1/8	1/8

   (b) Sharpen

0	−1	0
−1	5	−1
0	−1	0

   (c) Edge detection

0	−1	0
−1	4	−1
0	−1	0

   (d) Emboss

2	0	0
0	−1	0
0	0	−1

   Try this one on grayscale images.

   (e) Mean (averaging)

1/9	1/9	1/9
1/9	1/9	1/9
1/9	1/9	1/9

4. Implement a 3×3 filter of your choice. Describe the effect you intend it to produce.

This page intentionally left blank

Chapter 22

Image Synthesis

The **Mandelbrot set** is an interesting mathematical object that owes much of its popularity to the computer programs that create images of it.

Listing 22.1: Mandelbrot Set

```
# mandelbrot.py
from image import ImagePPM

def lerp(frac, low, high):
    return low + frac * (high - low)

def testpoint(c, maxreps):
    z = 0
    reps = 0
    while abs(z) < 2 and reps < maxreps:
        z = z ** 2 + c
        reps += 1
    frac = reps / maxreps
    return (0, 0, int(lerp(frac, 0, 255)))

def mandelbrot(xint, yint, size, maxreps):
    width, height = size
    img = ImagePPM.new(size)
    for i in range(width):
        for j in range(height):
            a = lerp(i / width, xint[0], xint[1])
            b = lerp(1 - j / height, yint[0], yint[1])
            c = complex(a, b)
            img.putpixel((i, j), testpoint(c, maxreps))
    return img

def main():
    img = mandelbrot((-2, 0.7), (-1.2, 1.2), (900, 800), 50)
    img.save("mbrot.ppm")

main()
```

$\Longrightarrow$ Caution: As you experiment with this program, some changes may cause it to take a long time to run.

Complex Numbers

The Mandelbrot set is defined using **complex numbers**, which have the form $a + bi$ where $i^2 = -1$. Complex numbers are visualized by plotting $z = a + bi$ as the point (a, b) in the usual plane $\mathbb{R}^2$.

Squaring a complex number $z = a + bi$ works like this:

$$z^2 = (a + bi)(a + bi) = a^2 + 2abi + b^2 i^2 = (a^2 - b^2) + (2ab)i$$

The size of a complex number is given by its absolute value, which is computed as

$$|\, z \,| = |\, a + bi \,| = \sqrt{a^2 + b^2}$$

The following process decides whether or not a complex number c is in the Mandelbrot set:

```
Begin with z = 0.
Repeat: replace z with z ** 2 + c.
If abs(z) <= 2 forever, then c is in the Mandelbrot set.
```

Programs cannot wait forever, so some upper limit must be set on the number of repetitions.

Complex Numbers in Python

Python supports complex numbers and complex arithmetic. We create a complex number with the built-in `complex()` function:

`complex(a, b)`	Complex number $a + bi$.

Then the arithmetic operations of addition, subtraction, multiplication, and exponentiation work as briefly described above. (Division is somewhat different, but we will not need it here.) The built-in absolute value function `abs()` also handles complex numbers correctly.

Parameter Tuples

The `mandelbrot()` function in Listing 22.1 requires quite a few parameters to specify the region to draw, the size of the resulting image, and the maximum number of repetitions to use when deciding whether or not a point is in the Mandelbrot set. By grouping these together into natural tuples, we make the parameter list manageable.

Linear Interpolation

There are several places in Listing 22.1 where we need to convert a fractional value in the interval $[0, 1]$ to some other range $[a, b]$. **Linear interpolation** is the name for performing this conversion in a linear way, so that, for example, if the fraction is $1/3$, then the interpolated value will be $1/3$ of the way from a to b. **Lerp** is an abbreviation of linear interpolation.

Exercises

22.1 Rewrite the body of `main()` in Listing 22.1 to use only one line of code.

22.2 Using the `lerp()` function from Listing 22.1, compute the following:

(a) `lerp(0, 1, 2)`

(b) `lerp(0.5, 3, 4)`

(c) `lerp(1, 2, 3)`

22.3 Use Listing 22.1 to:

(a) Explain the need for the `int()` conversion in line 14.

(b) Explain the need for "`1 -`" in line 22 instead of just `j / height`.

22.4 Use Listing 22.1 to:

(a) Give the region of the complex plane that is being drawn.

(b) Give the RGB color of points in the Mandelbrot set. Describe the color.

(c) Give the approximate RGB color of points that require very few repetitions to know that they are not in the set. Describe the color.

(d) Give the approximate RGB color of points that require almost the maximum number of repetitions to know that they are not in the set. Describe the color.

22.5 Describe and explain the effect of changing the `maxreps` parameter in Listing 22.1. How large a value are you able to use? How small?

22.6 Modify Listing 22.1 to reverse the color scheme of the original, so that points in the set are black and points not in the set are shades of blue.

22.7 Modify Listing 22.1 to separate out the color choice from the rest of the `testpoint()` function. Introduce a new function `color(reps, maxreps)` that computes the color based on `reps` and `maxreps`, and call this function at the end of `testpoint()`, after the number of `reps` has been determined.

22.8 Using the previous exercise, modify the `color()` function to highlight points that take many repetitions to decide that they are not in the Mandelbrot set. Set up three cases:

```
reps == maxreps:   black
0 <= frac < 0.5:   interpolate (0, 0, 100) to (0, 255, 100)
0.5 <= frac < 1.0: interpolate (0, 255, 100) to white
```

22.9 Modify Listing 22.1 to show these sections of the Mandelbrot set.

(a) Using x interval $[-1, -0.7]$ and y interval $[0.03, 0.3]$

(b) Using x interval $[-1, -0.7]$ and y interval $[-0.5, 0.5]$

Do either of the images appear distorted? Describe any distortion you see.

22.10 The **aspect ratio** of an image is the ratio of its width to its height. Use the images from the previous exercise to answer these questions about aspect ratio:

(a) Compute the aspect ratio of the region in the complex plane for part (a).

(b) Compute the aspect ratio of the region in the complex plane for part (b).

(c) Compute the aspect ratio of the images created, based on the number of pixels used.

Explain the distortion you found in Exercise 22.9.

22.11 It would be nice to have a function that automatically creates images with the correct aspect ratio. Let w and h be the actual width and height of the region being drawn. Then for any value s, if we let the width of the image (in pixels) be $s \cdot w$ and its height be $s \cdot h$, the aspect ratio of the image will be

$$\frac{sw}{sh} = \frac{w}{h},$$

which is the same as the region being drawn. Thus, s can be used to control the final size of the image.

(a) Write a function `imgsize(xint, yint, scale)` that returns an image size (width and height tuple) with the same aspect ratio as the given intervals and scaled according to `scale`.

(b) Use the `imgsize()` function to write a function

```
mandelbrotauto(xint, yint, scale, maxreps)
```

that draws the Mandelbrot set over the intervals `xint` and `yint` using the correct aspect ratio.

(c) Use the `mandelbrotauto()` function to correct the distortion from Exercise 22.9.

22.12 Use the previous exercise to write a function

```
mandelbrotcentered(center, size, maxreps)
```

that draws the region of the Mandelbrot set centered at `center` with width and height specified by the tuple `size`. (These are not the image width and height, but the size of the region being drawn in the complex plane.) Use your function to draw the region centered at $(-0.1, 1)$ with width and height 0.2.

22.13 Create your own interesting view of the Mandelbrot set.

22.14 **Julia sets** are defined similarly to the Mandelbrot set. The main difference is in how they treat z and c for the iteration of $f(z) = z^2 + c$. The Mandelbrot algorithm can be described as:

```
for every c in the region:
  test point c beginning with z = 0
```

In contrast, every c produces a different Julia set J_c. Given c, the Julia set algorithm for J_c is:

```
for every z in the region:
  test point z beginning the iteration with z itself
```

Note that c is still used in the iteration that replaces z with $z^2 + c$. Values of z that stay bounded in the iteration are in J_c.

Write a program to draw J_c for $c = -0.8 + 0.2i$ over the region with x-interval $[-1.6, 1.6]$ and y-interval $[-0.9, 0.9]$.

22.15 Draw Julia sets for values of c that are both in and not in the Mandelbrot set. Describe the differences that you see.

22.16 Create your own interesting image of a Julia set.

This page intentionally left blank

Chapter 23

Writing Classes

No programming language can anticipate all data types that a programmer might eventually need, and so object-oriented languages allow you to create new data types by defining classes. We start with a simple example to highlight the new concepts.

Listing 23.1: Sum of Dice

```
# dicesum.py

from random import randint

class Die:
    def __init__(self):
        self.roll()

    def roll(self):
        self.value = randint(1, 6)

    def __str__(self):
        return str(self.value)

def main():
    rolls = int(input("Enter the number of times to roll: "))
    die1 = Die()
    die2 = Die()
    counts = [0] * 13
    for i in range(rolls):
        die1.roll()
        die2.roll()
        counts[die1.value + die2.value] += 1
    for i in range(2, 13):
        print(str(i) + ": " + str(counts[i]))

main()
```

Object-Oriented Terminology

As we learn to create our own data types, it will help to have a clear understanding of the terminology:

Class A class is a template that defines objects of a new data type by specifying their state and behavior. For example, a `PlayingCard` class might define card objects for a game program.

Object An object is a specific **instance** of a class, with its own particular state. A program may create as many instances of a class as it needs. For example, a card game might create 52 `PlayingCard` objects to represent a deck.

State Objects have attributes or characteristics that are important to track for the purposes of a particular application. The current values of those attributes constitute the object's state. Object state is stored in **fields**, sometimes more specifically referred to as instance fields. A `PlayingCard` might have fields to store its rank and suit.

Behavior Objects will also have behaviors that need to be modeled by an application. Object behaviors are described by **methods**. For example, a `CardDeck` class might define a `shuffle()` method.

Methods are called **mutators** if they change the state of the object or **accessors** if they only return information about the state.

Constructor Constructors are called to create new objects. Each class will have its own constructor.

Remember that all data types in Python are object data types and so are defined by classes. What is different now is that we are writing classes to create new data types of our own.

Class Definitions

Classes are used to help manage the complexity that comes with building larger, more realistic programs. All class definitions begin with one line:

```
class <ClassName>:
    <body>
```

Class names are capitalized. In case you have been wondering, that is why we never capitalize function or variable names (except for all-caps constants). The body of a class definition contains all of its field and method definitions.

Field Definitions

Python is unusual in that it does not use explicit field declarations. In other words, you may have to search carefully in order to find all of the fields in a class definition. Fields are defined within the class by giving a value to a variable whose name begins with "`self.`" (notice the dot). That is, any variable whose name begins with "`self.`" is a field that is stored within the object. Every object has its own separate storage location for its fields, so that objects can hold different values in each of their fields.

Method Definitions

Methods are defined inside the body of a class in the same way as functions, using **`def`**, except that the first parameter for every method should be a variable named `self`. In other words, only two things distinguish an instance method from a regular function:

1. The **`def`** appears inside the body of a class.
2. The first parameter is `self`.

Method Calls Inside a Class

As you develop code for a class, you will often find it necessary for one instance method to call another instance method inside the class implementation. Python uses the same syntax for these method calls as it does for referencing an instance field: the method name must be prefaced with "`self.`" For example, the `__init__()` method of the `Die` class calls the `roll()` method on line 7 like this:

```
self.roll()
```

If you omit "`self.`," Python will assume that you are calling a regular function rather than an instance method.

Special Methods: `__init__()` and `__str()__`

In Python, methods with names that start and end with two underscores (__) will be called automatically in certain situations. Every class should define an `__init__()` method to give appropriate initial values to its instance fields. This is called **initializing** the object. In other words, `__init__()` is called automatically whenever the constructor creates a new object.

Most classes should also define a `__str__()` method, which is called any time the object appears in a `print()` or as an argument to `str()`. The `__str__()` method should return a string that describes the object. This doesn't necessarily mean including everything that is known about the object, but there should be enough information included to reasonably represent the object.

Calling Constructors

As mentioned above, constructors are used to create new objects. The syntax to call a constructor in Python is:

```
<ClassName>(<parameters>)
```

The constructor for a class always has the same name as the class itself. Any parameters sent to the constructor are passed on to the class's `__init__()` method.

When calling a constructor, we almost always immediately store the new object in a variable; otherwise, the object would reside in memory with no way to access it.

Exercises

23.1 Use Listing 23.1 to answer these questions:

(a) What state is stored in the `Die` class? Explain how you know.

(b) What behaviors are defined by the `Die` class? Explain how you know.

23.2 Use Listing 23.1 to answer these questions:

(a) Identify all instance fields in the `Die` class. Explain how you found them.

(b) Identify all instance methods in the `Die` class. Explain how you found them.

(c) For each method found in the previous question, indicate whether it is a mutator or accessor. Explain your answers.

(d) Identify all calls to the `Die()` constructor. How many instances are created, and where are the new objects stored?

(e) The `__init__()` method should "give appropriate initial values to its instance fields." Explain how the `__init__()` method of the `Die` class does this.

23.3 Explain the need for the calls to `str()` in line 25 of Listing 23.1.

23.4 Explain the use of the number "13" in lines 19 and 24 of Listing 23.1.

23.5 Rewrite Listing 23.1 to *not* use a `Die` class. Discuss the tradeoffs.

23.6 Modify Listing 23.1 to sum three dice instead of two.

23.7 Modify Listing 23.1 to sum n dice instead of two. Ask the user for the value of n.

23.8 Not all dice are cubes with 6 sides. There are dice for each of the platonic solids, as well as some other shapes. Modify the `Die` class to represent dice with different numbers of sides, and then modify Listing 23.1 to ask the user for the number of sides on the dice that will be simulated. Use a default value of 6 for the number of sides in the `Die` constructor.

23.9 Imagine writing a `Student` class for a registration system.

(a) List some elements that should be included in the `Student`'s state.

(b) List some possible behaviors that the `Student` object may need to perform.

23.10 Imagine writing a `Course` class for a registration system.

(a) List some elements that should be included in the `Course`'s state.

(b) List some possible behaviors that the `Course` object may need to perform.

23.11 Imagine writing an `Instructor` class for a registration system.

(a) List some elements that should be included in the `Instructor`'s state.

(b) List some possible behaviors that the `Instructor` object may need to perform.

23.12 Imagine writing an `Employee` class for a human resources system.

(a) List some elements that should be included in the `Employee`'s state.

(b) List some possible behaviors that the `Employee` object may need to perform.

23.13 Imagine writing a `Department` class for a human resources system.

(a) List some elements that should be included in the `Department`'s state.

(b) List some possible behaviors that the `Department` object may need to perform.

23.14 Write a `Circle` class that represents a circle. Store the radius of the circle, and include methods that return the circle's area and circumference. Write a `main()` function that creates a few circles and prints each area and circumference. (You do not need to write a `__str__()` method.)

23.15 Write a `Vector` class that represents a two-dimensional vector (x, y). Include a `__str__()` method, as well as a `norm()` method that returns the length of the vector, $\sqrt{x^2 + y^2}$. Write a `main()` function to test your code.

23.16 Write an `RGB` class that represents an RGB color. Store the red, green, and blue components. Include a `__str__()` method and a `luminance()` method that returns the luminance of the color. Write a `main()` function to test your code.

23.17 Write a `CD` class that represents a single music cd. Store the artist, title, genre, year of release, and playing time in minutes and seconds. However, instead of providing all of these as parameters to `__init__()`, write your class so that only the artist and title are passed to the constructor. Set the other fields to `None` initially, but provide `setyear()`, `setgenre()`, and `setplayingtime()` methods so that they can be set later. Have the `__str__()` method return the artist, title, and year, but only include the year if it was previously set. Write a `main()` function to test your code.

23.18 Write a `Card` class that represents a single playing card from a standard 52-card deck. Each card has a numeric rank and a suit that is hearts, diamonds, spades, or clubs. Include a `__str()__` method that returns a string representation of the card. Write a `main()` function that creates all 52 cards, shuffles them, and then prints the cards in shuffled order.

Chapter 24

Cooperating Classes

Classes are rarely used in isolation. Their power comes from interactions between objects, and choosing useful classes is the art of object-oriented design. The dice game Chuck-a-Luck provides a good example of a relatively simple game with enough complexity to warrant more than one class. Three dice are rolled, and the player needs to guess one of the numbers that will appear. If the number appears (either 1, 2, or 3 times), the player wins that many units; otherwise, the player loses one unit.

Listing 24.1: Chuck-a-Luck

```
# chuck.py

from dice import Dice

class ChuckALuck:
    def __init__(self):
        self.dice = Dice(3)
        self.score = 0

    def play(self):
        choice = int(input("Choose a number (0 to stop): "))
        while choice != 0:
            self.dice.rollall()
            print("The roll:", self.dice)
            matches = self.dice.count(choice)
            self.score += matches if matches > 0 else -1
            print("Current score:", self.score)
            choice = int(input("Choose a number (0 to stop): "))
        print("Thanks for playing.")

def main():
    game = ChuckALuck()
    game.play()

if __name__ == "__main__":
    main()
```

Focus on understanding the above code first, assuming that the `Dice` class does whatever it is supposed to do. Most of it should be familiar.

But where does the `Dice` class come from? It is defined in a separate `dice.py` file. For the first time, we have written one program that is divided into two separate files. Here is the second file:

Listing 24.2: Dice Classes

```
# dice.py

from random import randint

class Die:
    def __init__(self):
        self.roll()

    def roll(self):
        self.value = randint(1, 6)

    def __str__(self):
        return str(self.value)

    def __int__(self):
        return self.value

class Dice:
    def __init__(self, n=3):
        self.dice = [Die() for i in range(n)]

    def rollall(self):
        for die in self.dice:
            die.roll()

    def values(self):
        return list(map(int, self.dice))

    def count(self, value):
        return self.values().count(value)

    def __str__(self):
        return str(self.values())

def main():
    # Test
    dice = Dice(5)
```

```
38      print("Initial:", dice)
39      dice.rollall()
40      print("After roll:", dice)
41      print("Number of 2's:", dice.count(2))
42
43  if __name__ == "__main__":
44      main()
```

Most of this code should also be familiar.

Abstraction

Look again at Listing 24.1. By hiding all of the details of how dice work within the `Dice` class, we can focus on understanding the game itself. This is a form of **abstraction**, since we abstract away from the details involved in how the dice actually work. We are able to work at a higher abstraction level than if the code included all of those details in the middle of it.

Recall that in Chapter 3, functions were introduced as a "powerful tool of abstraction" (page 13). You should have a reasonable idea of what that means now: by isolating individual tasks in functions, we can ignore the details of those tasks and work at a higher level by simply calling the functions. Objects take this benefit of abstraction a step further. A single object can represent any amount of state and behavior, and allow us to think of it as a single unit.

Importing From Your Own Modules

When you write a class like `Die` or `Dice`, it would be nice to be able to write it once and then use it anytime you need to represent dice in a game. This is called **code reuse**. The way to reuse code in Python is to write the class definition(s) in a separate file, called a **module**, and then **import** the definitions you need from that module into whatever program wants to use them. Use the same syntax you have been using all semester:

```
from <module> import <name>
```

There is nothing special about a Python module file, except that it contains definitions you would like to use in other files.

Do not include "`.py`" in the name of the module in the **`import`** statement, but the name of the module file must end with "`.py`." It will also be simplest if you put the module file in the same folder or directory as the program that wants to use it.

```
if __name__ == "__main__":
```

Once you begin importing from your own modules, you will discover that any executable code in a module is run at the time it is imported. Take a look at the `main()` method in Listing 24.2. It is there to test the dice classes. The **if** statement on line 43 ensures that `main()` is only run when the dice module is the "main" module, rather than when it is imported. The built-in variable `__name__` is set to the string `"__main__"` only if the module is being run directly. When the dice module is imported, however, `__name__` is set to `"dice"`. Thus, testing the `__name__` variable allows us to determine whether or not the module is being run directly.

List Comprehensions

Line 20 of Listing 24.2 contains new syntax called a **list comprehension**. List comprehensions are a powerful and concise way to construct lists. In their simplest form, they look like this:

```
[<expr> <for <variable> in <sequence>]
```

This expression builds a list by evaluating `<expr>` for each iteration of the **for** loop. Thus, for example, the expression

```
[2 * i for i in range(5)]
```

evaluates to the list `[0, 2, 4, 6, 8]`.

List comprehensions may include additional **for** or **if** clauses. For example,

```
sqs = [x ** 2 for x in range(30) if x % 5 == 0]
```

only squares the `x` in `range(30)` that are multiples of 5.

Map

There is one final component of Listing 24.2 that is new. Both the `count()` method and the `__str__()` method need a list of integers rather than a list of dice, but the field `self.dice` is a list of `Die` objects.

The `__int__()` method of the `Die` class is similar to `__str__()`: it is called automatically any time the `int()` function is applied to a `Die`. Thus, to get a list of integer values of the dice, we need to apply the `int()` function to each element of the list `self.dice`. That is precisely what the built-in `map()` function does.

`map(f, items)`	Apply `f` to each item in `items`.

The `map()` function returns an **iterator**, which may be converted to a list using the `list()` type converter as on line 27.

Calls to `map()` that are intended to construct a list may always be replaced with an equivalent list comprehension:

```
[f(item) for item in items]
```

produces the same list as

```
list(map(f, items))
```

The choice is usually based on personal preference or which feels simpler in the given context.

Exercises

24.1 Use Listing 24.1 to answer these questions:

(a) Is the "3" necessary in line 7? Explain why or why not.

(b) Explain line 16.

(c) Explain the **`if`** statement at line 25.

(d) List all fields and methods of the `ChuckALuck` class.

24.2 Explain how the `__str__()` method of the `Dice` class in Listing 24.2 works. In particular, compare its behavior with changing line 33 to just return `str(self.dice)`.

24.3 Add the statement

```
print(__name__)
```

just before the **`if`** statement at line 43 of Listing 24.2. Report what is printed when this module is run directly, then report what is printed when Listing 24.1 is run.

24.4 Rewrite the `Dice` class initializer to *not* use list comprehension. Discuss the tradeoffs.

24.5 Add these methods to the `Dice` class:

(a) `roll(i)` to roll only the `i`-th die

(b) `sumall()` to return the sum of all of the dice

Test your code.

24.6 Use the `Dice` class from Listing 24.1 to rewrite Exercise 23.7.

24.7 Rewrite these exercises using list comprehension:

(a) Exercise 14.7

(b) Exercise 14.9

24.8 Rewrite these exercises using list comprehensions with **`if`** clauses:

(a) Exercise 14.15

(b) Exercise 14.16

(c) Exercise 14.17

24.9 Modify Listing 24.2 to use an equivalent list comprehension instead of the call to `map()`.

24.10 Write a program `craps.py` to play the dice game Craps. Use appropriate classes.

24.11 Using the `PlayingCard` class from Exercise 23.18, write a `Deck` class to represent a deck of 52 playing cards. Include a `shuffle()` method and a `__str__()` method. Test your code.

24.12 Write a program `war.py` to simulate playing the card game War. Since there is no strategy involved, have the program play both hands and collect statistics about the winner, number of rounds, etc. Use appropriate classes.

Case Study: PPM Image Class

Now that you have seen and written a few classes, it may be helpful to see the complete code for the `ImagePPM` class used in Chapters 21 and 22.

Listing 24.3: ImagePPM

```
# image.py

def clamp(x, a, b):
    return max(a, min(b, x))

class ImagePPM:
    @classmethod
    def new(cls, size):
        newimg = ImagePPM()
        newimg.size = size
        newimg.maxval = 255
        newimg.data = [0] * 3 * size[0] * size[1]
        return newimg

    @classmethod
    def open(cls, fname):
        newimg = ImagePPM()
        with open(fname) as f:
            header = f.readline()
            if not header.startswith("P3"):
                return None
            nextline = f.readline()
            while nextline.startswith("#"):
                nextline = f.readline()
            newimg.size = tuple(map(int, nextline.split()))
            newimg.maxval = int(f.readline())
            newimg.data = list(map(int, f.read().split()))
        return newimg

    def index(self, coords):
        i, j = coords
        return 3 * (j * self.size[0] + i)
```

```
33
34      def getpixel(self, coords):
35          k = self.index(coords)
36          return tuple(self.data[k:k + 3])
37
38      def putpixel(self, coords, color):
39          k = self.index(coords)
40          for i in range(3):
41              self.data[k + i] = clamp(color[i], 0, 255)
42
43      def save(self, fname):
44          width, height = self.size
45          with open(fname, "w") as f:
46              f.write("P3\n")
47              f.write(str(width) + " " + str(height) + "\n")
48              f.write(str(self.maxval) + "\n")
49              for i in range(3 * width * height):
50                  f.write(str(self.data[i]) + "\n")
```

The main new feature is the use of class methods instead of explicit constructors to create new image objects.

Class Methods

To this point, the term "method" has been used to refer to object methods, also known as instance methods. Recall that an object method is always called from an object:

```
<object>.<method>(<arguments>)
```

A **class method** is similar except that it is usually called from the class rather than a particular object:

```
<ClassName>.<method>(<arguments>)
```

Class methods are defined with a first parameter of `cls` (short for "class," which is already a keyword) instead of `self`, as in lines 8 and 16. The main thing to remember when writing a class method is that it does not have access to any instance variables via `self`. Class methods in Python are prefaced with the **decorator** `@classmethod`. The decorator indicates to the interpreter our intention to use these methods as class methods.

The two class methods `open()` and `new()` in Listing 24.3 are examples of **factory** methods because they create and return new `ImagePPM` objects.

PPM Image Formats

There are actually two types of PPM images: ASCII and raw binary.

ASCII PPM images look like this:

```
P3
# CREATOR: GIMP PNM Filter Version 1.1
300 225
255
89
157
232
...
```

The first line must be "`P3`" and indicates the ASCII PPM format. Optional comment lines, beginning with a "`#`", may follow the P3. The next line contains two integers: the image width and height (300 and 225 in this case). The following line contains the maximum color value for each RGB component, which is usually 255. All remaining lines contain RGB pixel data. In this case, the first pixel has color (89, 157, 232).

Because they are stored in ASCII format, ASCII PPM images may be viewed in a text editor, which is handy when tracking down bugs. However, it also means that ASCII PPM files are huge: every character requires a separate byte of storage, and no compression has been used.

Raw binary PPM images, while still uncompressed, are more efficient than ASCII PPM. They have the same header format except that the type is "`P6`." However, the RGB color data is stored in binary, using one byte per component instead of one byte per character. Two bytes per component are used if the maximum color value is 65535. Because only the header lines are in ASCII, raw binary files cannot be viewed in a text editor.

Exercises

1. Measure the difference in size between an ASCII PPM and a raw binary PPM of the same image.

2. Explain in your own words how each of these methods works from Listing 24.3:

   (a) `new()`

   (b) `open()`

   (c) `index()`

(d) `getpixel()`

(e) `putpixel()`

(f) `save()`

3. Modify Listing 24.3 to add a `.copy()` method to the `ImagePPM` class that returns an exact copy of `self`.

Chapter 25

Related Classes

Classes may also be related to each other through a mechanism called inheritance. When one class inherits from another, it inherits all the state and behavior of the original class, and then has the opportunity to define additional new state and behavior.

The **Sierpinski triangle** is an interesting mathematical object (a **fractal**, like the Mandelbrot set) that is constructed by repeatedly removing the middle from a solid triangle.

Listing 25.1: Sierpinski Triangle

```
# sierpinski.py

from turtle import Turtle, setworldcoordinates, exitonclick

class SierpinskiTriangle(Turtle):
    size = 2
    def __init__(self, n, x, y):
        Turtle.__init__(self, visible=False)
        self.n = n
        self.speed(0)
        self.penup()
        self.goto(x, y)

    def draw(self):
        if self.n == 0:
            self.begin_fill()
            for i in range(3):
                self.forward(SierpinskiTriangle.size)
                self.left(120)
            self.end_fill()
        else:
            for i in range(3):
                SierpinskiTriangle(self.n - 1,
                                   self.xcor(),
                                   self.ycor()).draw()
                self.forward(SierpinskiTriangle.size ** self.n)
```

```
27                  self.left(120)
28
29  def main():
30      setworldcoordinates(0, 0, 75, 75)
31      SierpinskiTriangle(5, 5, 5).draw()
32      exitonclick()
33
34  if __name__ == "__main__":
35      main()
```

If this program takes too long to complete, try reducing the first parameter to the constructor in `main()`.

The `Turtle` Class

The `turtle` module provides both procedural and object-oriented versions of its functionality. Part I only used procedural code (Python functions) because we had not used objects at that point in the course. Listing 25.1 introduces object-oriented `turtle` module code via the `Turtle` class. The `Turtle` class defines turtle objects that draw and move using method calls that are exactly the same as the function calls we used earlier. In fact, the function calls are implemented in the background by calling the corresponding method on a single internal `Turtle` object. One advantage of using `Turtle` objects explicitly is that your program can then work with more than one turtle at a time.

Some of the methods available on a `Turtle` object `t` are:

`t.pendown()`	Start drawing the turtle's path.
`t.penup()`	Stop drawing the turtle's path.
`t.dot()`	Draw a dot at current location.
`t.goto(x, y)`	Move to position (x, y).
`t.forward(n)`	Move forward `n` units.
`t.left(theta)`	Turn left angle `theta` (default degrees).
`t.xcor()`	Current position's x coordinate.
`t.ycor()`	Current position's y coordinate.
`t.speed(s)`	Set speed (1=slow to 10=fast; 0 is "instantly").
`t.begin_fill()`	Turn filling on.
`t.end_fill()`	Stop filling.

You can find the description of other methods in the Python library documentation.

Inheritance

When we define a new class using **inheritance**, the new class inherits all of the state and behavior of the original class, known as the **base class**. The new class is called an **extension** of the base class, and it may define additional new state and behavior. Base classes are also known as **parent** classes or **superclasses**; extensions are also known as **child** classes or **subclasses**.

Inheritance is said to model an "**is-a**" relationship because an instance of the extension *is an* instance of the base class.

The syntax to define a class as an extension of a base class in Python is:

```
class <ClassName>(<BaseClass>):
    <body>
```

Thus, in Listing 25.1, the `SierpinskiTriangle` class extends the `Turtle` base class. Because of this, all of the methods defined in the `Turtle` class are available to be called on any `SierpinskiTriangle` object. We say that a `SierpinskiTriangle` object "is a" `Turtle`. In addition, `SierpinskiTriangle` objects have additional state and behavior that `Turtle` objects do not.

Superclass Initializer

If you define an `.__init__()` method for a class extension, it is usually a good idea to call the base class's `.__init__()` method first to make sure that the state defined for the base class is initialized to reasonable values before trying to set additional state for the extension. The syntax to make this call is:

```
<BaseClass>.__init__(self, <other parameters>)
```

Class Variables

Recall that objects store data in instance variables (fields) and that the syntax to access a particular object's field is `<object>.<field>`. Every object has its own storage for instance variables and thus different objects may store different values in their fields.

Occasionally, it is helpful for a class to store data that is shared among all instances of the class; in this case, a **class variable** is appropriate. For example, in Listing 25.1, the `size` variable of the `SierpinskiTriangle` class is shared by all instances of that class and controls the size of the triangles that are drawn.

There is no special syntax for declaring class variables in Python. Instead, any variable that is assigned a value *inside* of a class definition but *outside* all

method definitions of the class (as on line 6) is automatically a class variable. The syntax to access the value of a class variable is:

```
<ClassName>.<variable>
```

This is analogous to using `<object>.<variable>` to access an instance variable.

Class variables are public and may be accessed by code outside of the class using the same syntax.

Exercises

25.1 Use the `SierpinskiTriangle` class from Listing 25.1 to answer these questions:

(a) List the new state and behavior that `SierpinskiTriangle` adds to `Turtle`.

(b) Explain why the `size` variable is better as a class variable than an instance variable.

(c) Explain why `.penup()` is called in the initializer without any later call to `.pendown()`.

(d) Describe as specifically as you can where all drawing is done in this program.

25.2 Use Listing 25.1 to answer these questions:

(a) Determine the total number of turtle objects that are created.

(b) Determine the starting position and orientation of a new turtle that is initialized by `Turtle.__init__()`.

(c) One of the three vertices is used as the starting point for drawing each triangle. Which vertex does this code use?

25.3 Modify Listing 25.1 to draw the lower-right triangle first.

25.4 Modify Listing 25.1 to use a `Screen` object instead of direct function calls. Use the Python documentation if you need it.

25.5 Modify Listing 25.1 to ask the user for the initial value of `n`. Adjust the screen coordinates so that the image fits the screen appropriately.

25.6 Write a program to implement the following algorithm that uses any three noncolinear points in the plane A, B, and C:

```
Set p randomly to one of A, B, or C.
Repeat:
    Set q randomly to one of A, B, or C.
    Set p to the midpoint between p and q.
    Put a dot at p.
```

Describe the result. You do not need to write any classes.

25.7 Implement the previous exercise using a class that inherits from `Turtle`.

25.8 Write a program `dicepoker.py` to play dice poker. The game is played by rolling five dice. The user then chooses to either stay or re-roll some of the dice. After two options to re-roll or the player stays, the hand is evaluated. Scoring hands are five of a kind, a straight, four of a kind, a full house, three of a kind, or two pair. Give each of these an appropriate value, and keep track of the amount of money held by the player. Use appropriate classes.

25.9 Write a program `pig.py` to play the dice game Pig. Turns alternate between two players. During a player's turn, the player rolls one die repeatedly until either a 1 is rolled or the player decides to stand. If a 1 appears at any time, the turn is over and the player scores 0. Otherwise, when a player stands, the turn is over and the player receives the sum of all of his or her rolls during that turn. The first player to reach 100 wins. Design appropriate classes.

This page intentionally left blank

Chapter 26

Functional Programming

While primarily thought of and designed as a scripting language, Python supports several different programming paradigms. This chapter provides an introduction to functional programming by revisiting a problem from Part II.

Listing 26.1: Word Frequency (Functional Version)

```
# wordfreqfp.py

import string
import itertools

def removepunc(s):
    return s.translate(s.maketrans("", "", string.punctuation))

def getwords(fname):
    with open(fname) as f:
        return removepunc(f.read().lower()).split()

def frequency(words):
    return list(map(lambda xy: (xy[0], len(list(xy[1]))),
                    itertools.groupby(sorted(words))))

def main():
    print(frequency(getwords("moby.txt")))

main()
```

Compare this code with Listing 18.1: the style should feel different. Like Listing 18.1, this program should be run from the command line. In fact, you may need to replace the `print()` in `main()` with this loop if you receive an out-of-memory error:

```
for word, freq in frequency(getwords("moby.txt")):
    print(word, freq)
```

Functional programs generally do not use **for** loops, but it may be necessary here in order to get output. The `map()` function is described in Chapter 24.

Programming Paradigms

Python supports four of the most common programming paradigms:

Imperative Programming Sequences of statements are used to change the program state. State is usually maintained and updated with assignment statements.

Procedural Programming A form of imperative programming in which procedures (functions in Python) are used to organize tasks and subtasks.

Object-Oriented Programming Objects manage state by communicating with each other using method calls.

Functional Programming Emphasizes functions that transform input to output without side effects that modify program state. In particular, assignment statements (which change program state) are avoided.

Most of the code we have written so far has been procedural and therefore imperative. The past few chapters emphasized object-oriented programming, and in this chapter we explore functional programming. As you begin to study Listing 26.1, notice how it consists almost entirely of function calls. There is no dictionary or other data structure that separately stores the word frequencies. Instead, all of the data is passed from one function to the next until the program is finished.

In fact, you have already seen some features of Python that are functional in style:

- List comprehensions
- Using `map()`
- Passing functions as parameters
- The `applyfn()` function from Listing 21.1

Python demonstrates that the lines between these paradigms do not have to be strict.

Program Overview

Because the functional programming style emphasizes functional transformations of input to output, we begin with an overview of the transformations that take place in Listing 26.1.

File
↓
List of words

↓
Sorted list of words
↓
Grouped list of `(word, [word, word, ...])` pairs
↓
Output list of `(word, count)` pairs

Each of these steps is accomplished by calling one or more Python functions. Exercise 26.2 asks you to identify the function(s) used at each step.

Iterators and Iterables

Before getting further into the details of Listing 26.1, it will be helpful to understand iterators and iterables a bit better. Recall from Chapter 4 that the `range()` function returns an iterable, and in Chapter 24 we saw that the `map()` function returns an iterator. Neither of these returns an explicit list.

Think of an iterator as a "potential list" that will provide its elements, one by one, when asked for them. An iterable is closely related and is any object that can provide an iterator. The distinction between iterators and iterables is not crucial for us; what matters is that many list operations will in fact work with any iterable or iterator. You may have noticed this already in the documentation. And any time you need the entire list from an iterator or iterable, just use the `list()` type converter. Line 14 of Listing 26.1 uses `list()` twice in this way.

Grouping List Elements

Once the list of words has been sorted, we want to group together all words that are the same so that the counts can be totaled at the next stage. The `itertools` module provides the `groupby()` function for exactly this purpose:

`itertools.groupby(items)`	Group `items` into pairs: `(key, [item, item, ...])`.

The list sent to `groupby()` should be sorted first so that all identical items are collected together. The return value of `groupby()` is an iterator that contains pairs:

```
(key, <iterator of items with the same key>)
```

where the second element of each pair is itself an iterator of all the items in the original list that match the same key. Thus, in Listing 26.1, because we are sending a list of words to `groupby()` and the keys are the words themselves, the output at this stage for a word that occurs three times (such as "wheelbarrow") will be:

```
("wheelbarrow", <iterator for 3 occurrences of "wheelbarrow">)
```

An iterator cannot be asked directly for its length, but it can be converted to a list first, and then the length of the list may be computed. That is the approach taken on line 14 of Listing 26.1.

Lambda Expressions

Lambda expressions are useful for quick, one-time function definitions that are not worth writing a separate function for. They create **anonymous** functions, that is, functions that do not have a name. The syntax is:

```
lambda <parameters> : <expression>
```

The body of the lambda form is limited to be a single expression rather than a complete function body.

A lambda definition is equivalent to:

```
def name(<parameters>):
    return <expression>
```

except that the function has no name. Lambda expressions are especially useful with `map()`, as in Listing 26.1.

Listing 26.1 requires a lambda form that takes a tuple as a parameter, because `itertools.groupby()` returns an iterator of tuples. In line 14, the tuple is named `xy`, and the components of the tuple are then accessed using indices. Again, the second item in each tuple, `xy[1]`, is an iterator, so the lambda expression converts it to a list before computing its length.

Recursion

Rather than using **`while`** or **`for`** loops, functional programs often implement repetition through recursion. A function is **recursive** if it calls itself, usually on a smaller argument. The sequence of calls must eventually terminate in a **base case** that does not generate a new recursive call.

For example, the factorial function implemented with recursion might look like this:

```
def factorial(n):
    return n * factorial(n - 1) if n > 1 else 1
```

Here, the smaller argument in the recursive call is $n - 1$, and the base case is when $n \leq 1$. No state is explicitly being kept in a variable, as it would be in an iterative version.

The Sierpinski triangle is a recursive object, and the `.draw()` method of Listing 25.1 uses a form of recursion as it calls itself on three new triangle objects.

Exercises

26.1 Describe the differences between Listing 18.1 and 26.1 in your own words. Discuss the tradeoffs between the two approaches.

26.2 Identify the Python function(s) used at each stage of the Program Overview diagram on page 176.

26.3 Use Listing 26.1 to answer these questions:

(a) Describe the effect of the lambda expression in the `frequency()` function.

(b) Rewrite `frequency()` to used a named function instead of a lambda expression. Give the function an appropriate name. Discuss the tradeoffs.

26.4 Rewrite the `frequency()` function in Listing 26.1 to use a list comprehension instead of `map()` and **`lambda`**.

26.5 Modify Listing 21.1 to use a lambda expression instead of `f()`.

26.6 Rewrite the `harmonic()` function from Listing 4.1 in a functional style. Hint: `sum()` works with iterables of numbers.

26.7 Rewrite each of these functions from Listing 8.1 in a functional style:

(a) `smallestdivisor()`. You may want a separate function to compute all divisors of `n` larger than 1.

(b) `main()`. Write a new function `primes(n)` that returns a list of all primes less than `n`.

26.8 Rewrite your implementation of Listing 8.2 in a functional style. It may help to introduce a helper function `mysqrtguess(guess, k)` that `mysqrt()` calls with an initial guess to start the process. Then you can write `mysqrtguess(guess, k)` recursively rather than using a loop.

26.9 Rewrite Listing 10.1 to use a lambda expression instead of `f()`.

26.10 Rewrite the `estimate_area()` function from Listing 10.1 in a functional style. Introduce new functions if you need them.

26.11 Rewrite each of these functions from Listing 12.1 in a functional style:

(a) `complement()`

(b) `random_dna()`

26.12 Rewrite each of these functions from Listing 15.1 in a functional style:

(a) `signature()`

(b) `matches()`

Chapter 27

Parallel Programming

Real-world problems often take too long to solve on a single processor because of their size or computational demands. **Parallel processing** is a family of mechanisms for coordinating multiple tasks that work in parallel to solve a single problem. In this chapter, we explore the use of parallel programming with the example from Chapter 10.

Listing 27.1: Monte Carlo Integration (Parallel Version)

```
# montecarlo.py

from random import uniform
from math import exp
import multiprocessing

def count_hits(f, a, b, m, n):
    hits = 0
    for i in range(n):
        x = uniform(a, b)
        y = uniform(0, m)
        if y <= f(x):
            hits += 1
    return hits

def estimate_area_mp(f, a, b, m, n=1000):
    workers = multiprocessing.cpu_count()
    pool = multiprocessing.Pool(workers)
    total = m * (b - a)
    x = [a + i * (b - a) / workers for i in range(workers + 1)]
    hits = []
    for i in range(workers):
        pool.apply_async(count_hits,
                         (f, x[i], x[i + 1], m, n // workers),
                         callback=hits.append)
    pool.close()
    pool.join()
    return sum(hits) * total / n

```

```
30 def f(x):
31     return exp(-x ** 2)
32
33 def main():
34     print(estimate_area_mp(f, 0, 2, 1))
35
36 if __name__ == "__main__":
37     main()
```

This example may need to be run from the command line.

Multiprocessing

Multiprocessing refers generally to computing that uses more than one CPU within a single system. These systems can be built within one enclosure, such as a older **supercomputers**, or distributed across many separate boxes in a **cluster**.

As manufacturers have found it increasingly difficult to build faster CPUs, they have turned to **multi-core processors**, combining several processors onto a single chip, in order to obtain competitive advantages. This widespread use of multiple processors in desktops, laptops, and tablets makes parallel computing relevant to everyone, not just specialists.

Python Multiprocessing Pools

The Python `multiprocessing` module provides several ways of tapping into the parallelism of a multi-core processor. Listing 27.1 creates a **pool** of **worker processes** that can be assigned tasks. In this case, the number of workers is set to be the number of CPU cores.

`multiprocessing.Pool(n)`	Pool with `n` workers.
`multiprocessing.cpu_count()`	Number of CPU cores in system.

These worker processes are relatively **heavyweight**, meaning that each one requires a fair number of resources and code to manage their work. Thus, these pools are best used with relatively small numbers of workers.

The tasks assigned to workers are called **asynchronously**, meaning that the program does not wait for the call to return before moving on to the next line of code. (All of the other function calls we have made have been **synchronous**, where the function call has needed to complete before moving on to the next step in the program.) Asynchronous calls are necessary for work to be done in parallel: if all calls are synchronous, then only one task is done at a time. When an asynchronous call completes, the function designated as the **callback** is immediately run.

The following method initiates an asynchronous call from a multiprocessing pool:

`pool.apply_async(f, args, callback=g)`	Call `f` asynchronously.

The `args` parameter must be a tuple containing all arguments for the call to `f`. When the call to `f` completes, its return value is passed to the callback `g`.

If several asynchronous calls have been initiated, the main program needs some way to know when all of the calls have completed. Listing 27.1 uses these two method calls:

`pool.close()`	Indicate no more tasks will be given to pool.
`pool.join()`	Wait for all workers to finish.

The `.close()` method must be called before `.join()`.

Exercises

27.1 Use Listing 27.1 to answer these questions:

(a) Give the number of workers that are created on your system.

(b) Give the line number where this program waits for all of the workers to finish their tasks.

(c) Describe how the work is divided in this program between different workers.

(d) Discuss other possible ways of dividing the Monte Carlo simulation between different workers.

(e) Describe what the callback function does and how it contributes to the solution of the main problem.

27.2 Research the reason that a lambda expression cannot be used in Listing 27.1 instead of `f()`.

27.3 Research the reason that lightweight **threads** do not benefit from multiple cores in Python in the same way that heavyweight processes do.

27.4 Compare the performance of Listing 27.1 with Listing 10.1. Discuss your results.

27.5 Measure the performance of Listing 27.1 with different numbers of workers. Find an optimal number of workers for your system with this problem. Discuss the results.

27.6 Apply the techniques of this chapter to a different problem that may benefit from this type of parallelism. Measure program performance and discuss the results.

27.7 Research the pool `.map()` method and:

(a) Describe why it is possible for `.map()` to benefit from multiple cores, whereas we needed to use the asynchronous version of `.apply()` to benefit from multiple cores in that case.

(b) Apply the pool `.map()` method to a functional program that could benefit from parallelism.

Chapter 28

Graphical User Interfaces

Most software written today uses a **graphical user interface (GUI)** rather than the text-based interface we have used in most of our examples. In this chapter, we rewrite the Chuck-a-Luck program from Chapter 24 to use a graphical interface.

Listing 28.1: Chuck-a-Luck (GUI Version)

```
# chuckgui.py

from dicegui import DiceGUI
from tkinter import *

class ChuckALuckGUI(Frame):
    def __init__(self, parent):
        Frame.__init__(self, parent)
        parent.geometry("500x300")
        parent.title("Chuck-A-Luck")
        self.dice = DiceGUI(3, self)
        self.dice.pack(fill="y", expand=True)
        self.score = 0
        self.makecontrols()
        self.makescore()
        self.pack(fill="y", expand=True)

    def makecontrols(self):
        self.choice = IntVar()
        controlframe = Frame(self)
        for i in range(1, 7):
            Radiobutton(controlframe, text=str(i), value=i,
                        variable=self.choice).pack(side="left")
        Label(controlframe, width=5).pack(side="left")
        Button(controlframe, width=10, text="Roll",
               command=self.roll).pack(side="left")
        controlframe.pack()

    def makescore(self):
        self.scorestr = StringVar()
```

```
        Label(self, textvariable=self.scorestr,
              font=("Helvetica", 24)).pack(fill="y", expand=True)
        self.updatescore()

    def roll(self):
        self.dice.rollall()
        matches = self.dice.count(self.choice.get())
        self.score += matches if matches > 0 else -1
        self.updatescore()

    def updatescore(self):
        self.scorestr.set("Score: " + str(self.score))

root = Tk()
app = ChuckALuckGUI(root)
root.mainloop()
```

Be aware that the GUI window created by this program may be hidden behind other windows when you run it. We also need GUI code for the dice.

Listing 28.2: Dice GUI

```
# dicegui.py

from dice import Dice
from tkinter import *

class DiceGUI(Frame, Dice):
    def __init__(self, n, parent):
        Frame.__init__(self, parent)
        Dice.__init__(self, n)
        self.loadimages()
        self.n = n
        self.labels = [Label(self) for i in range(self.n)]
        for i in range(self.n):
            self.setimage(i)
            self.labels[i].pack(side="left")

    def loadimages(self):
        self.images = [None] * 7
        for i in range(1, 7):
            fname = "die" + str(i) + ".ppm"
            self.images[i] = PhotoImage(file=fname)

    def rollall(self):
```

```
24          Dice.rollall(self)
25          for i in range(self.n):
26              self.setimage(i)
27
28      def setimage(self, i):
29          self.labels[i].config(image=self.images[int(self.dice[i])])
```

The tkinter Module

The `tkinter` module is one of the standard libraries for developing graphical applications in Python. It provides an interface to **Tk**, which is a cross-platform GUI library for the **Tcl** language. Because a typical GUI application uses many components from `tkinter`, the `*`-form of the **`import`** is common.

The `Tk` class from the `tkinter` module creates the main window for a GUI application. Once a `Tk` object has been created, then we just add GUI components (called "widgets") to it and finish with a call to the `Tk` object's `.mainloop()` method.

Event-Driven Programming

In order to understand the `.mainloop()` method, we need to discuss **event-driven programming**. A GUI application needs to respond to user **events** such as mouse-clicks, pushing buttons, and choosing menu items. In order to do this, the program goes into an **event loop**:

```
while True:
    # get next event
    # process event
```

Given a `Tk` object named `root`, these two methods control the event loop:

`root.mainloop()`	Start main event loop.
`root.quit()`	Stop main event loop.

The event loop is also halted if the user closes the main window of the application.

Window Methods

Listing 28.1 uses two other methods that are available on all `tkinter` windows, including the root `Tk` object:

`win.geometry("widthxheight")`	Set size of window.
`win.title(str)`	Set window title to `str`.

Widgets

GUI components in `tkinter` are known as **widgets**. Listings 28.1 and 28.2 use the following widget classes:

`Frame`	Group widgets for layout.
`Label`	Display text or image.
`Button`	Single button.
`Radiobutton`	Button allowing only one from a group to be selected at a time. Buttons with same `variable` form a group.

The first parameter to a widget's constructor is always the GUI component that will contain the widget, known as its **parent**. The new widget is then a **child** of the parent. Other parameters are assigned values with **keyword arguments**, which have the form

```
keyword=value
```

The advantage of using keywords in this case is that the caller does not need to list every parameter in the precisely correct order. Instead, only those parameters being used are specified, and they may be listed in any order.

Widget Options

Most of the keyword arguments used to set widget options in Listings 28.1 and 28.2 are self-explanatory. For example, the `command` parameter of the `Button` widget specifies the command to execute when the button is clicked. Check the `tkinter` documentation if you have questions and to see additional options.

All of the options that can be set via keyword arguments in widget constructors may also be set later with the `.config()` method:

`widget.config(keyword=value, ...)`	Set widget options.

or by setting dictionary values:

`widget["keyword"]=value`	Set widget option.

The keyword must be in quotes when using the dictionary syntax.

IntVar and StringVar

The `tkinter` module provides its own special variable classes to use with those widgets that require variables in order to communicate data either to or from the rest of a Python program. For example, the set of `Radiobutton` objects uses the variable `self.choice` to inform the `.roll()` method which radio button was chosen. An ordinary Python variable cannot be used for this purpose; instead, an `IntVar` is created. Similarly, a `StringVar` is used by the score `Label` to update and display the score.

If `v` is either an `IntVar` or `StringVar`, these methods are used to access or modify its contents:

`v.get()`	Get value of `v`.
`v.set(value)`	Set `v` to `value`.

Geometry Managers

GUI applications need some way to describe and define how their components should be organized onscreen. The `tkinter` module provides three different **geometry managers** for this purpose: the **Pack** manager organizes components into groups, the **Grid** manager places them in "row, column" positions, and the **Place** manager allows precise pixel locations. The Pack manager and Grid manager are probably used the most often; Listing 28.1 uses the Pack manager for its layout.

$\Longrightarrow$ Caution: Never use more than one geometry manager in a single window.

The basic idea of the Pack manager is that each component calls its `.pack()` method to describe how it should be placed within its parent.

`widget.pack(keyword=value, ...)`	Add `widget` to its parent with keyword options.

Some of the `.pack()` method options are:

side="left", "right", "top", or "bottom" Pack this widget into its parent by sliding it up against the specified side. Default is top.

fill=None, "x", "y", or "both" If `None`, the widget keeps its original size. Filling in the x or y directions (or both) means the widget will use all of the space given to it by the parent in the specified direction(s). Default is `None`.

expand=False or True If `True`, this widget is willing to expand to take extra space available in the parent. Default is `False`.

Frames are often used with the Pack manager to group objects together in the layout. Generally, one set of options is used to pack widgets into a `Frame`, and then the `Frame` specifies how it should be packed into its parent.

$\Longrightarrow$ Caution: When using the Pack manager, components will not appear until you call their `.pack()` method.

Multiple Inheritance

Take a look at line 6 of Listing 28.2: it appears that the `DiceGUI` class inherits from both `Frame` and `Dice`. This mechanism is called **multiple inheritance**, and it does exactly what you expect: the `DiceGUI` class inherits state and behavior from both the `Frame` and `Dice` classes. A `DiceGUI` instance *is a* `Frame` and a set of `Dice`.

Notice a couple of details in the code: first, in the `__init__()` method, both superclass `__init__()` methods are called. Secondly, the `DiceGUI` class **overrides** the definition of `rollall()` it inherited from `Dice`. The old version is called as part of the new definition in line 24:

```
Dice.rollall(self)
```

This allows the `DiceGUI` class to extend the behavior of the original method.

PhotoImages

Finally, the `tkinter` module allows using GIF or raw binary PPM images inside GUI applications with the `PhotoImage` class:

`PhotoImage(file=fname)`	Open image in `fname`.

Inside a `tkinter` program, `PhotoImage` objects may be used anywhere a widget expects an image.

In Listing 28.2, a `PhotoImage` is created for each face of the die and stored in the `images` field of the `DiceGUI` class. Then a call to the `Label .config()` method allows us to update the image any time the dice have been rolled.

Exercises

28.1 Use Listings 28.1 and 28.2 to answer these questions:

(a) List all instance variables in the `ChuckALuckGUI` class. Do not include those it inherits from `Frame`.

(b) List all instance variables in the `DiceGUI` class. Do include those inherited from the `Dice` class (in Listing 23.1), but do not include those from `Frame`.

28.2 Use Listings 28.1 and 28.2 to answer these questions:

(a) Explain how the radio buttons get the correct labels.

(b) Find and explain the line of code that checks how many dice have the same value as the selected radio button.

(c) Find and explain the line of code that updates the image of a die after it has been rolled.

28.3 Describe the purpose of each of these widgets in the `makecontrols()` method of Listing 28.1:

(a) `controlframe`

(b) The `Label` in line 24

28.4 Describe and explain the result of commenting out each of these lines of code from Listing 28.1:

(a) Line 12

(b) Line 16

(c) Line 27

28.5 Use Listing 28.2 to answer these questions:

(a) Explain the need for the assignment in line 18. Hint: what happens if that line is removed?

(b) Explain why line 18 uses the value 7 instead of 6.

(c) Consider alternative ways of writing the `loadimages()` method. Discuss the tradeoffs.

28.6 Modify Listing 28.1 to turn off all of the `expand` options. Describe and then explain the result.

28.7 Python offers the option of defining an **alias** for a module with

```
import <module> as <alias>
```

Modify Listing 28.1 to use the alias `tk` for `tkinter` instead of importing all names from the `tkinter` module. Discuss the tradeoffs. (You do not need to modify Listing 28.2.)

28.8 Instead of making `ChuckALuckGUI` a subclass of `Frame`, an alternative design is to create a `Frame` object in `__init()__` and store it in an instance variable of `ChuckALuckGUI`. This technique is called using **composition** instead of inheritance. Composition models a "**has-a**" relationship because, in this case, the application class *has a* `Frame`. Rewrite Listing 28.1 with this design. Discuss the tradeoffs.

28.9 Modify Listings 28.1 and 28.2 to display the dice in a vertical column on the right side of the window. Arrange the other components in a column on the left side of the window. Make whatever changes are necessary to produce a nice layout.

28.10 Research the `tkinter` Grid geometry manager and modify Listings 28.1 and 28.2 to use it instead of the Pack manager. Reproduce the layout of the original program as closely as possible.

28.11 Modify Listing 28.1 to improve the (very minimal) design so that it is more interesting and appealing.

28.12 Design and implement a GUI version of dice poker (see Exercise 25.8).

28.13 Design and implement a GUI application of your choice.

Bibliography

[1] Thomas Akenine-Möller and Eric Haines. *Real-Time Rendering*. A. K. Peters, Ltd., Natick, MA, 2nd edition, 2002.

[2] G. Evans, J. Blackledge, and P. Yardley. *Numerical Methods for Partial Differential Equations*. Springer-Verlag, Berlin, 2000.

[3] Brian W. Kernighan and P. J. Plauger. *Software Tools in Pascal*. Addison-Wesley, Reading, MA, 1981.

[4] Dan E. Krane and Michael L. Raymer. *Fundamental Concepts of Bioinformatics*. Benjamin Cummings, San Francisco, 2003.

[5] Claude Shannon. A mathematical theory of communication. *Bell System Technical Journal*, 27:379–423, 623–656, July, October 1948.

[6] James E. Zull. *The Art of Changing the Brain*. Stylus Publishing, LLC, Sterling, VA, 2002.

This page intentionally left blank

Computer Science

Suitable for newcomers to computer science, **A Concise Introduction to Programming in Python** provides a succinct, yet complete, first course in computer science using the Python programming language.

The book features:

- Short, modular chapters with brief and precise explanations, intended for one class period
- Early introduction of basic procedural constructs such as functions, selection, and repetition, allowing them to be used throughout the course
- Objects are introduced in the middle of the course, and class design comes toward the end
- Examples, exercises, and projects from a wide range of application domains, including biology, physics, images, sound, mathematics, games, and textual analysis
- No external libraries are required, simplifying the book's use in common lab spaces

Each chapter introduces a main idea through a concrete example and a series of exercises. Designed to teach programming in a concise, yet comprehensive way, this book provides a timely introduction for students and anyone interested in learning Python.

CRC Press
Taylor & Francis Group
an **informa** business
www.crcpress.com

6000 Broken Sound Parkway, NW
Suite 300, Boca Raton, FL 33487
711 Third Avenue
New York, NY 10017
2 Park Square, Milton Park
Abingdon, Oxon OX14 4RN, UK

K14279
ISBN: 978-1-4398-9694-5
90000
9 781439 896945