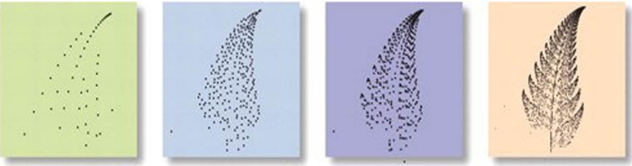
INTRODUCTION TO

Programming in Python



An Interdisciplinary Approach

Robert Sedgewick • Kevin Wayne • Robert Dondero

Introduction to Programming in Python

Introduction to Programming in Python

An Interdisciplinary Approach

Robert Sedgewick
Kevin Wayne
Robert Dondero

Princeton University

Addison-Wesley

New York • Boston • Indianapolis • San Francisco
Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

For information about buying this title in bulk quantities, or for special sales opportunities (which may include electronic versions; custom cover designs; and content particular to your business, training goals, marketing focus, or branding interests), please contact our corporate sales department at corpsales@pearsoned.com or (800) 382-3419.

For government sales inquiries, please contact governmentsales@pearsoned.com.

For questions about sales outside the United States, please contact international@pearsoned.com.

Visit us on the Web: informit.com/aw

Library of Cataloging-in-Publication Data
Sedgewick, Robert, 1946-
Introduction to programming in Python : an interdisciplinary approach / Robert Sedgewick, Kevin Wayne, Robert Dondero.
pages cm
Includes indexes.
ISBN 978-0-13-407643-0 (hardcover : alk. paper)—ISBN 0-13-407643-5
1. Python (Computer program language) 2. Computer programming. I. Wayne, Kevin Daniel, 1971- II. Dondero, Robert. III. Title.
QA76.73.P98S43 2015
005.13'3—dc23
2015011936

ISBN-13: 978-0-13-407643-0
ISBN-10: 0-13-407643-5

Text printed in the United States on recycled paper at Edwards Brothers Malloy in Ann Arbor, Michigan.
First printing, June 2015

To Adam, Andrew, Brett, Robbie,

Henry, Iona, Rose, Peter,

and especially Linda

To Jackie and Alex

To my family,

especially Ellen and Meghan

This page intentionally left blank

Contents

This page intentionally left blank

Programs

Elements of Programming

Functions and Modules

Object-Oriented Programming

Data Types

Creating Data Types

Designing Data Types

Case Study: N-Body Simulation

Algorithms and Data Structures

Performance

Sorting and Searching

Stacks and Queues

Symbol Tables

Case Study: Small-World Phenomenon

This page intentionally left blank

This page intentionally left blank

Preface

THE BASIS FOR EDUCATION IN THE last millennium was "reading, writing, and arithmetic"; now it is reading, writing, and *computing*. Learning to program is an essential part of the education of every student in the sciences and engineering. Beyond direct applications, it is the first step in understanding the nature of computer science's undeniable impact on the modern world. This book aims to teach programming to those who need or want to learn it, in a scientific context.

Our primary goal is to *empower* students by supplying the experience and basic tools necessary to use computation effectively. Our approach is to teach students that composing a program is a natural, satisfying, and creative experience. We progressively introduce essential concepts, embrace classic applications from applied mathematics and the sciences to illustrate the concepts, and provide opportunities for students to write programs to solve engaging problems.

We use the Python programming language for all of the programs in this book—we refer to "Python" after "programming in the title to emphasize the idea that the book is about *fundamental concepts in programming*, not Python per se. This book teaches basic skills for computational problem solving that are applicable in many modern computing environments, and is a self-contained treatment intended for people with no previous experience in programming.

This book is an *interdisciplinary* approach to the traditional CS1 curriculum, in that we highlight the role of computing in other disciplines, from materials science to genomics to astrophysics to network systems. This approach emphasizes for students the essential idea that mathematics, science, engineering, and computing are intertwined in the modern world. While it is a CS1 textbook designed for any first-year college student interested in mathematics, science, or engineering, the book also can be used for self-study or as a supplement in a course that integrates programming with another field.

Coverage The book is organized around four stages of learning to program: basic elements, functions, object-oriented programming, and algorithms . We provide the basic information readers need to build confidence in composing programs at each level before moving to the next level. An essential feature of our approach is to use example programs that solve intriguing problems, supported with exercises ranging from self-study drills to challenging problems that call for creative solutions.

Basic elements include variables, assignment statements, built-in types of data, flow of control , arrays, and input/output, including graphics and sound.

Functions and modules are the student's first exposure to modular programming. We build upon familiarity with mathematical functions to introduce Python functions, and then consider the implications of programming with functions, including libraries of functions and recursion. We stress the fundamental idea of dividing a program into components that can be independently debugged, maintained, and reused.

Object-oriented programming is our introduction to data abstraction. We emphasize the concepts of a data type and their implementation using Python's class mechanism. We teach students how to *use*, *create*, and *design* data types. Modularity, encapsulation, and other modern programming paradigms are the central concepts of this stage.

Algorithms and data structures combine these modern programming paradigms with classic methods of organizing and processing data that remain effective for modern applications. We provide an introduction to classical algorithms for sorting and searching as well as fundamental data structures and their application, emphasizing the use of the scientific method to understand performance characteristics of implementations.

Applications in science and engineering are a key feature of the text. We motivate each programming concept that we address by examining its impact on specific applications. We draw examples from applied mathematics, the physical and biological sciences, and computer science itself, and include simulation of physical systems, numerical methods, data visualization, sound synthesis, image processing, financial simulation, and information technology. Specific examples include a treatment in the first chapter of Markov chains for web page ranks and case studies that address the percolation problem, *n*-body simulation, and the small-world phenomenon. These applications are an integral part of the text. They engage students in the material, illustrate the importance of the programming concepts, and provide persuasive evidence of the critical role played by computation in modern science and engineering.

Our primary goal is to teach the specific mechanisms and skills that are needed to develop effective solutions to any programming problem. We work with complete Python programs and encourage readers to use them. We focus on programming by individuals, not programming in the large.

Use in the Curriculum This book is intended for a first-year college course aimed at teaching novices to program in the context of scientific applications. Taught from this book, prospective majors in any area of science and engineering will learn to program in a familiar context. Students completing a course based on this book will be well prepared to apply their skills in later courses in science and engineering and to recognize when further education in computer science might be beneficial.

Prospective computer science majors, in particular, can benefit from learning to program in the context of scientific applications. A computer scientist needs the same basic background in the scientific method and the same exposure to the role of computation in science as does a biologist, an engineer, or a physicist.

Indeed, our interdisciplinary approach enables colleges and universities to teach prospective computer science majors and prospective majors in other fields of science and engineering in the *same* course. We cover the material prescribed by CS1, but our focus on applications brings life to the concepts and motivates students to learn them. Our interdisciplinary approach exposes students to problems in many different disciplines, helping them to choose a major more wisely.

Whatever the specific mechanism, the use of this book is best positioned early in the curriculum. First, this positioning allows us to leverage familiar material in high school mathematics and science. Second, students who learn to program early in their college curriculum will then be able to use computers more effectively when moving on to courses in their specialty. Like reading and writing, programming is certain to be an essential skill for any scientist or engineer. Students who have grasped the concepts in this book will continually develop that skill through a lifetime, reaping the benefits of exploiting computation to solve or to better understand the problems and projects that arise in their chosen field.

Prerequisites This book is suitable for typical science and engineering students in their first year of college. That is, we do not expect preparation beyond what is typically required for other entry-level science and mathematics courses.

Mathematical maturity is important. While we do not dwell on mathematical material, we do refer to the mathematics curriculum that students have taken in high school, including algebra, geometry, and trigonometry. Most students in our target audience automatically meet these requirements. Indeed, we take advantage of their familiarity with the basic curriculum to introduce basic programming concepts.

Scientific curiosity is also an essential ingredient. Science and engineering students bring with them a sense of fascination with the ability of scientific inquiry to help explain what goes on in nature. We leverage this predilection with examples of simple programs that speak volumes about the natural world. We do not assume any specific knowledge beyond that provided by typical high school courses in mathematics, physics, biology, or chemistry.

Programming experience is not necessary, but also is not harmful. Teaching programming is our primary goal, so we assume no prior programming experience. But composing a program to solve a new problem is a challenging intellectual task, so students who have written numerous programs in high school can benefit from taking an introductory programming course based on this book . The book can support teaching students with varying backgrounds because the applications appeal to both novices and experts alike.

Experience using a computer is not necessary, but also is not at all a problem. College students use computers regularly, to communicate with friends and relatives, listen to music, to process photos, and as part of many other activities. The realization that they can harness the power of their own computer in interesting and important ways is an exciting and lasting lesson.

In summary, virtually all students in science and engineering fields are prepared to take a course based on this book as a part of their first-semester curriculum.

Goals What can *instructors* of upper-level courses in science and engineering expect of students who have completed a course based on this book?

We cover the CS1 curriculum, but anyone who has taught an introductory programming course knows that expectations of instructors in later courses are typically high: each instructor expects all students to be familiar with the computing environment and approach that he or she wants to use. A physics professor might expect some students to design a program over the weekend to run a simulation; an engineering professor might expect other students to be using a particular package to numerically solve differential equations; or a computer science professor might expect knowledge of the details of a particular programming environment. Is it realistic to meet such diverse expectations? Should there be a different introductory course for each set of students?

Colleges and universities have been wrestling with such questions since computers came into widespread use in the latter part of the 20th century. Our answer to them is found in this common introductory treatment of programming, which is analogous to commonly accepted introductory courses in mathematics, physics, biology, and chemistry. *An Introduction to Programming in Python* strives to provide the basic preparation needed by all students in science and engineering, while sending the clear message that there is much more to understand about computer science than programming. Instructors teaching students who have studied from this book can expect that they will have the knowledge and experience necessary to enable those students to adapt to new computational environments and to effectively exploit computers in diverse applications.

What can *students* who have completed a course based on this book expect to accomplish in later courses?

Our message is that programming is not difficult to learn and that harnessing the power of the computer is rewarding. Students who master the material in this book are prepared to address computational challenges wherever they might appear later in their careers. They learn that modern programming environments, such as the one provided by Python, help open the door to any computational problem they might encounter later, and they gain the confidence to learn, evaluate, and use other computational tools. Students interested in computer science will be well prepared to pursue that interest; students in science and engineering will be ready to integrate computation into their studies.

Booksite An extensive amount of information that supplements this text may be found on the web at

```
http://introcs.cs.princeton.edu/python
```

For economy, we refer to this site as the *booksite* throughout. It contains material for instructors, students, and casual readers of the book. We briefly describe this material here, though, as all web users know, it is best surveyed by browsing. With a few exceptions to support testing, the material is all publicly available.

One of the most important implications of the booksite is that it empowers instructors and students to use their own computers to teach and learn the material. Anyone with a computer and a browser can begin learning to program by following a few instructions on the booksite. The process is no more difficult than downloading a media player or a song. As with any website, our booksite is continually evolving. It is an essential resource for everyone who owns this book. In particular, the supplemental materials are critical to our goal of making computer science an integral component of the education of all scientists and engineers.

For *instructors*, the booksite contains information about teaching. This information is primarily organized around a teaching style that we have developed over the past decade, where we offer two lectures per week to a large audience, supplemented by two class sessions per week where students meet in small groups with instructors or teaching assistants. The booksite has presentation slides for the lectures, which set the tone.

For *teaching assistants*, the booksite contains detailed problem sets and programming projects, which are based on exercises from the book but contain much more detail. Each programming assignment is intended to teach a relevant concept in the context of an interesting application while presenting an inviting and engaging challenge to each student. The progression of assignments embodies our approach to teaching programming. The booksite fully specifies all the assignments and provides detailed, structured information to help students complete them in the allotted time, including descriptions of suggested approaches and outlines for what should be taught in class sessions.

For *students*, the booksite contains quick access to much of the material in the book, including source code, plus extra material to encourage self-learning. Solutions are provided for many of the book's exercises, including complete program code and test data. There is a wealth of information associated with programming assignments, including suggested approaches, checklists, FAQs, and test data.

For *casual readers* , the booksite is a resource for accessing all manner of extra information associated with the book's content. All of the booksite content provides web links and other routes to pursue more information about the topic under consideration. There is far more information accessible than any individual could fully digest, but our goal is to provide enough to whet any reader's appetite for more information about the book's content.

Acknowledgments This project has been under development since 1992, so far too many people have contributed to its success for us to acknowledge them all here. Special thanks are due to Anne Rogers, for helping to start the ball rolling; to Dave Hanson, Andrew Appel, and Chris van Wyk, for their patience in explaining data abstraction; and to Lisa Worthington, for being the first to truly relish the challenge of teaching this material to first-year students. We also gratefully acknowledge the efforts of `/dev/126` ; the faculty, graduate students, and teaching staff who have dedicated themselves to teaching this material over the past 25 years here at Princeton University; and the thousands of undergraduates who have dedicated themselves to learning it.

Robert Sedgewick
Kevin Wayne
Robert Dondero

April 2015

2.1 Defining Functions

You have been composing code that calls Python functions since the beginning of this book, from writing strings with `stdio.writeln()` to using type conversion functions such as `str()` and `int()` to computing mathematical functions such as `math.sqrt()` to using all of the functions in `stdio`, `stddraw`, and `stdaudio`. In this section, you will learn how to define and call your own functions.

In mathematics, a function maps an input value of one type (the domain) to an output value of another type (the range). For example, the square function $f(x) = x^2$ maps 2 to 4, 3 to 9, 4 to 16, and so forth. At first, we work with Python functions that implement mathematical functions, because they are so familiar. Many standard mathematical functions are implemented in Python's `math` module, but scientists and engineers work with a broad variety of mathematical functions, which cannot all be included in the module. At the beginning of this section, you will learn how to implement and use such functions on your own.

Later, you will learn that we can do more with Python functions than implement mathematical functions: Python functions can have strings and other types as their domain or range, and they can have side effects such as writing output. We also consider in this section how to use Python functions to organize programs and thereby simplify complicated programming tasks.

From this point forward, we use the generic term *function* to mean either *Python function* or *mathematical function* depending on the context. We use the more specific terminology only when the context requires that we do so.

Functions support a key concept that will pervade your approach to programming from this point forward: *Whenever you can clearly separate tasks within a computation, you should do so.* We will be overemphasizing this point throughout this section and reinforcing it throughout the rest of the chapter (and the rest of the book). When you write an essay, you break it up into paragraphs; when you compose a program, you break it up into functions. Separating a larger task into smaller ones is much more important when programming than when writing an essay, because it greatly facilitates debugging, maintenance, and reuse, which are all critical in developing good software.

Using and defining functions As you know from the functions you have been using, the effect of calling a Python function is easy to understand. For example, when you place `math.sqrt(a-b)` in a program, the effect is as if you had replaced that code with the *return value* that is produced by Python's `math.sqrt()` function when passed the expression `a-b` as an *argument*. This usage is so intuitive that we have hardly needed to comment on it. If you think about what the system has to do to create this effect, however, you will see that it involves changing a program's *control flow.* The implications of being able to change the control flow in this way are as profound as doing so for conditionals and loops.

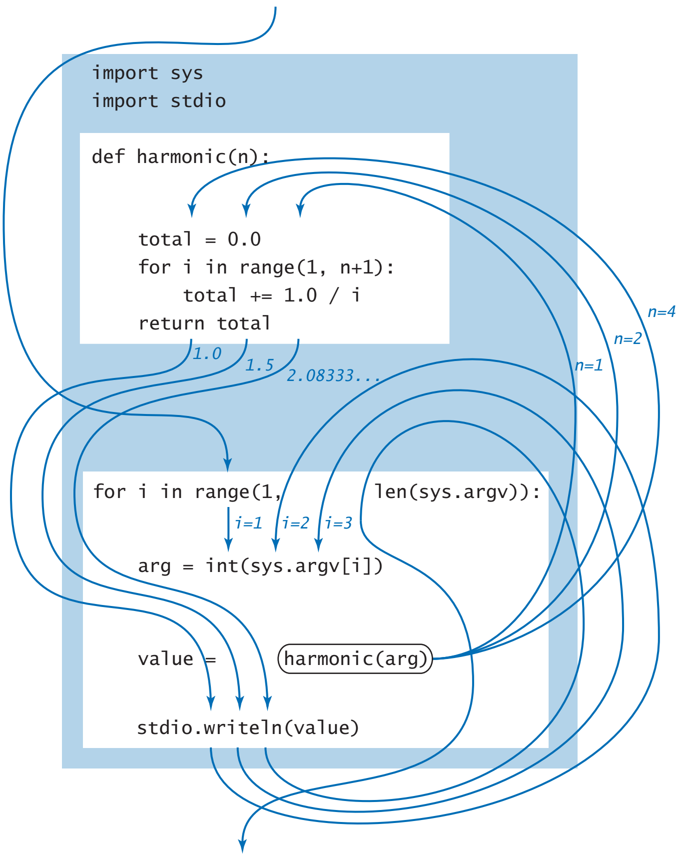
You can define functions in any Python program, using the `def` statement that specifies the function signature, followed by a sequence of statements that constitute the function. We will consider the details shortly, but begin with a simple example that illustrates how functions affect control flow. Our first example, Program 2.1.1 (`harmonicf.py`), includes a function named `harmonic()` that takes an argument `n` and computes the `n`th harmonic number (see Program 1.3.5). It also illustrates the typical structure of a Python program, having three components:

- A sequence of `import` statements
- A sequence of *function definitions*
- Arbitrary *global code*, or the body of the program

Program 2.1.1 has two `import` statements, one function definition, and four lines of arbitrary global code. Python executes the global code when we invoke the program by typing **`python harmonicf.py`** on the command line; that global code calls the `harmonic()` function defined earlier.

The implementation in `harmonicf.py` is preferable to our original implementation for computing harmonic numbers (Program 1.3.5) because it clearly separates the two primary tasks performed by the program: calculating harmonic numbers and interacting with the user. (For purposes of illustration, we have made the user-interaction part of the program a bit more complicated than in Program 1.3.5.) *Whenever you can clearly separate tasks within a computation, you should do so*. Next, we carefully examine precisely how `harmonicf.py` achieves this goal.

Control flow. The diagram on the next page illustrates the flow of control for the command **`python harmonicf.py 1 2 3`**. First, Python processes the `import` statements, thus making all of the features defined in the `sys` and `stdio` modules available to the program. Next, Python processes the definition of the `harmonic()` function at lines 4 through 8, but *does not execute the function*—Python executes a function only when it is called. Then, Python executes the first statement in the global code after the function definition, the `for` statement, which proceeds normally until Python begins to execute the statement `value = harmonic(arg)`, starting by evaluating the expression `harmonic(arg)` when `arg` is `1`. To do so it transfers control to the `harmonic()` function—the flow of control passes to the code in the function definition. Python initializes the "parameter" variable `n` to `1` and the "local" variable `total` to `0.0` and then executes the `for` loop within `harmonic()`, which terminates after one iteration with `total` equal to `1.0`. Then, Python executes the `return` statement at the end of the definition of `harmonic()`, causing the flow of control to jump back to the calling statement `value = harmonic(arg)`, continuing from where it left off, but now with the expression `harmonic(arg)` replaced by `1.0`. Thus, Python assigns `1.0` to `value` and writes it to standard output. Then, Python iterates the loop once more, and calls the `harmonic()` function a second time with `n` initialized to `2`, which results in `1.5` being written. The process is then repeated a third time with `arg` (and then `n`) equal to `4`, which results in `2.0833333333333333` being written. Finally, the `for` loop terminates and the whole process is complete. As the diagram indicates, the simple code masks a rather intricate flow of control.



Flow of control for **`python harmonicf.py 1 2 4`**

Program 2.1.1 Harmonic numbers (revisited) (`harmonicf.py`)

```
import sys
import stdio

def harmonic(n):
    total = 0.0
    for i in range(1, n+1):
        total += 1.0 / i
    return total

for i in range(1, len(sys.argv)):
    arg = int(sys.argv[i])
    value = harmonic(arg)
    stdio.writeln(value)
```

`n`	*parameter variable*
`i`	*loop index*
`total`	*return value*

`i`	*argument index*
`arg`	*argument*
`value`	*Harmonic number*

This program writes to standard output the harmonic numbers specified as command-line arguments. The program defines a function `harmonic()` *that, given an* `int` *argument* n*, computes the* n*th harmonic number* $1 + 1/2 + 1/3 + \ldots + 1/n$.

```
% python harmonicf.py 1 2 4
1.0
1.5
2.083333333333333
```

```
% python harmonicf.py 10 100 1000 10000
2.9289682539682538
5.187377517639621
7.485470860550343
9.787606036044348
```

Abbreviation alert. We continue to use the abbreviations that we introduced in Section 1.2 for functions and function calls. For example, we might say, "The function call `harmonic(2)` returns the value `1.5`," instead of the more accurate but verbose "When we pass to `harmonic()` a reference to an object of type `int` whose value is `2`, it returns a reference to an object of type `float` whose value is `1.5`." We strive to use language that is succinct and only as precise as necessary in a given context.

Informal function call/return trace. One simple approach to following the control flow through function calls is to imagine that each function writes its name and argument(s) when it is called and its return value just before returning, with indentation added on calls and subtracted on returns. The result enhances the process of tracing a program by writing the values of its variables, which we have been using since Section 1.2. An informal trace for our example is shown at right. The added indentation exposes the flow of the control, and helps us check that each function has the effect that we expect. Generally, adding calls on `stdio.writef()` to trace *any* program's control flow in this way is a fine approach to begin to understand what it is doing. If the return values match our expectations, we need not trace the function code in detail, saving us a substantial amount of work.

```
i = 1
arg = 1
harmonic(1)
    total = 0.0
    total = 1.0
    return 1.0
value = 1.0
i = 2
arg = 2
harmonic(2)
    total = 0.0
    total = 1.0
    total = 1.5
    return 1.5
value = 1.5
i = 3
arg = 4
harmonic(4)
    total = 0.0
    total = 1.0
    total = 1.5
    total = 1.8333333333333333
    total = 2.083333333333333
    return 2.083333333333333
value = 2.083333333333333
```

Informal trace with function call/return for ***python harmonicf.py 1 2 4***

For the rest of this chapter, your programming will be centered on creating and using functions, so it is worthwhile to consider in more detail their basic properties and, in particular, the terminology surrounding functions. Following that, we will study several examples of function implementations and applications.

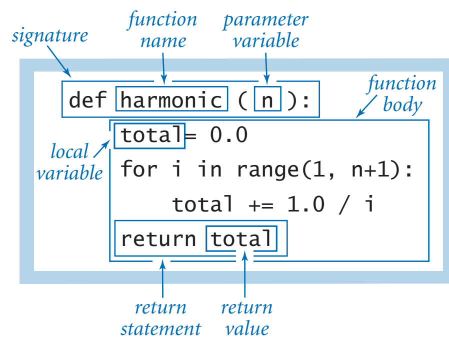
Basic terminology. As we have been doing throughout, it is useful to draw a distinction between abstract concepts and Python mechanisms to implement them (the Python `if` statement implements the conditional, the `while` statement implements the loop, and so forth). There are several concepts rolled up in the idea of a mathematical function and there are Python constructs corresponding to each, as summarized in the table at the top of the following page. While you can rest assured that these formalisms have served mathematicians well for centuries (and have served programmers well for decades), we will refrain from considering in detail all of the implications of this correspondence and focus on those that will help you learn to program.

When we use a symbolic name in a formula that defines a mathematical function (such as $f(x) = 1 + x + x^2$), the symbol x is a placeholder for some input value

concept	*Python construct*	*description*
function	function	mapping
input value	argument	input to function
output value	return value	output of function
formula	function body	function definition
independent variable	parameter variable	symbolic placeholder for input value

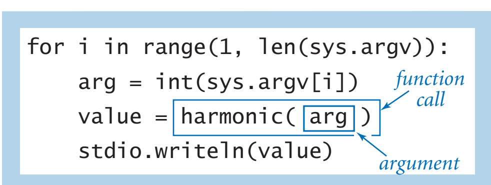
that will be substituted into the formula to determine the output value. In Python, we use a *parameter variable* as a symbolic placeholder and we refer to a particular input value where the function is to be evaluated as an *argument*.

Function definition. The first line of a function definition, known as its *signature*, gives a name to the function and to each parameter variable. The signature consists of the keyword `def`; the *function name*; a sequence of zero or more parameter variable names separated by commas and enclosed in parentheses; and a colon. The indented statements following the signature define the *function body*. The function body can consist of the kinds of statements that we discussed in Chapter 1. It also can contain a *return statement*, which transfers control back to the point where the function was called and returns the result of the computation or *return value*. The body may also define *local variables*, which are variables that are available only inside the function in which they are defined.



Anatomy of a function definition

Function calls. As we have seen throughout, a Python function call is nothing more than the function name followed by its arguments, separated by commas and enclosed in parentheses, in precisely the same form as is customary for mathematical functions. As noted in Section 1.2, each argument can be an expression, which is evaluated and the resulting value passed as input to the function. When the function finishes, the return value takes the place of the function call as if it were the value of a variable (perhaps within an expression).



Anatomy of a function call

Multiple arguments. Like a mathematical function, a Python function can have more than one parameter variable, so it can be called with more than one argument. The function signature lists the name of each parameter variable, separated by commas. For example, the following function computes the length of the hypotenuse of a right triangle with sides of length `a` and `b`:

```
def hypot(a, b)
    return math.sqrt(a*a + b*b)
```

Multiple functions. You can define as many functions as you want in a `.py` file. The functions are independent, except that they may refer to each other through calls. They can appear in any order in the file:

```
def square(x):
    return x*x

def hypot(a, b):
    return math.sqrt(square(a) + square(b))
```

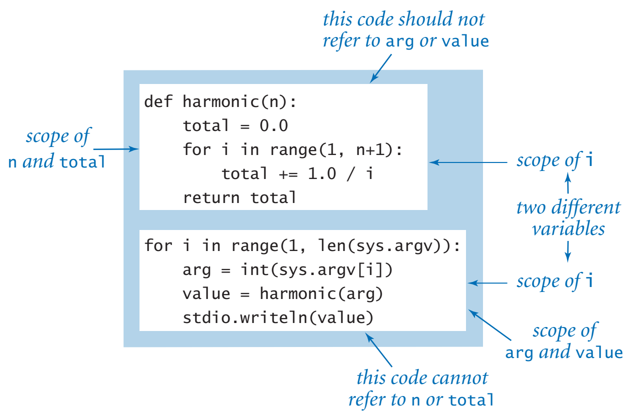
However, the definition of a function must appear before any global code that calls it. That is the reason that a typical Python program contains (1) `import` statements, (2) function definitions, and (3) arbitrary global code, in that order.

Multiple return statements. You can put `return` statements in a function wherever you need them: control goes back to the calling program as soon as the first `return` statement is reached. This *primality-testing* function is an example of a function that is natural to define using multiple `return` statements:

```
def isPrime(n):
    if n < 2: return False
    i = 2
    while i*i <= n:
        if n % i == 0: return False
        i += 1
    return True
```

Single return value. A Python function provides only one return value to the caller (or, more precisely, it returns a reference to one object). This policy is not as restrictive as it might seem, because Python data types can contain more information than a single number, boolean, or string. For example, you will see later in this section that you can use arrays as return values.

Scope. The *scope* of a variable is the set of statements that can refer to that variable directly. The scope of a function's local and parameter variables is limited to that function; the scope of a variable defined in global code—known as a *global variable*—is limited to the `.py` file containing that variable. Therefore, global code cannot refer to either a function's local or parameter variables. Nor can one function refer to either the local or parameter variables that are defined in another function. When a function defines a local (or parameter) variable with the same name as a global variable (such as `i` in Program 2.1.1), the variable name in the function refers to the local (or parameter) variable, not the global variable.



Scope of local and parameter variables

A guiding principle when designing software is to define each variable so that its scope is as small as possible. One of the important reasons that we use functions is so that changes made to one part of a program will not affect an unrelated part of the program. So, while code in a function *can* refer to global variables, it *should not* do so: all communication from a caller to a function should take place via the function's parameter variables, and all communication from a function to its caller should take place via the function's return value. In Section 2.2, we consider a technique for removing most global code, thereby limiting scope and the potential for unexpected interactions.

Default arguments. A Python function may designate an argument to be *optional* by specifying a *default value* for that argument. If you omit an optional argument in a function call, then Python substitutes the default value for that argument. We have already encountered a few examples of this feature. For example, `math.log(x, b)` returns the base-`b` logarithm of `x`. If you omit the second argument, then `b` defaults to `math.e`—that is, `math.log(x)` returns the natural logarithm of `x`. It might appear that the `math` module has two different logarithm functions, but it actually has just one, with an optional argument and a default value.

You can specify an optional argument with a default value in a user-defined function by putting an equals sign followed by the default value after the parameter variable in the function signature. You can specify more than one optional argument in a function signature, but all of the optional arguments must follow all of the mandatory arguments.

For example, consider the problem of computing the nth *generalized harmonic number of order* r: $H_{n,\,r} = 1 + 1/2^r + 1/3^r + \ldots + 1/n^r$. For example, $H_{1,\,2} = 1$, $H_{2,\,2} = 5/4$, and $H_{2,\,2} = 49/36$. The generalized harmonic numbers are closely related to the Riemann zeta function from number theory. Note that the nth generalized harmonic number of order $r = 1$ is equal to the nth harmonic number. Therefore it is appropriate to use `1` as the default value for `r` if the caller omits the second argument. We specify by writing `r=1` in the signature:

```
def harmonic(n, r=1):
    total = 0.0
    for i in range(1, n+1):
        total += 1.0 / (i ** r)
    return total
```

With this definition, `harmonic(2, 2)` returns `1.25`, while both `harmonic(2, 1)` and `harmonic(2)` return `1.5`. To the client, it appears that we have two different functions, one with a single argument and one with two arguments, but we achieve this effect with a single implementation.

Side effects. In mathematics, a function maps one or more input values to some output value. In computer programming, many functions fit that same model: they accept one or more arguments, and their only purpose is to return a value. A *pure function* is a function that, given the same arguments, always return the same value, without producing any observable *side effects*, such as consuming input, producing output, or otherwise changing the state of the system. So far, in this section we have considered only pure functions.

However, in computer programming it is also useful to define functions that do produce side effects. In fact, we often define functions whose only purpose is to produce side effects. An explicit `return` statement is optional in such a function: control returns to the caller after Python executes the function's last statement. Functions with no specified return value actually return the special value `None`, which is usually ignored.

For example, the `stdio.write()` function has the side effect of writing the given argument to standard output (and has no specified return value). Similarly, the following function has the side effect of drawing a triangle to standard drawing (and has no specified return value):

```
def drawTriangle(x0, y0, x1, y1, x2, y2):
    stddraw.line(x0, y0, x1, y1)
    stddraw.line(x1, y1, x2, y2)
    stddraw.line(x2, y2, x0, y0)
```

It is generally poor style to compose a function that both produces side effects and returns a value. One notable exception arises in functions that read input. For example, the `stdio.readInt()` function both returns a value (an integer) and produces a side effect (consuming one integer from standard input).

Type checking. In mathematics, the definition of a function specifies both the domain and the range. For example, for the harmonic numbers, the domain is the positive integers and the range is the positive real numbers. In Python, we do not specify the types of the parameter variables or the type of the return value. As long as Python can apply all of the operations within a function, Python executes the function and returns a value.

If Python cannot apply an operation to a given object because it is of the wrong type, it raises a run-time error to indicate the invalid type. For example, if you call the `square()` function defined earlier with an `int` argument, the result is an `int`; if you call it with a `float` argument, the result is a `float`. However, if you call it with a string argument, then Python raises a `TypeError` at run time.

This flexibility is a popular feature of Python (known as *polymorphism*) because it allows us to define a single function for use with objects of different types. It can also lead to unexpected errors when we call a function with arguments of unanticipated types. In principle, we could include code to check for such errors, and we could carefully specify which types of data each function is supposed to work with. Like most Python programmers, we refrain from doing so. However, in this book, our message is that *you should always be aware of the type of your data*, and the functions that we consider in this book are built in line with this philosophy, which admittedly clashes with Python's tendency toward polymorphism. We will discuss this issue in some detail in SECTION 3.3.

The table below summarizes our discussion by collecting together the function definitions that we have examined so far. To check your understanding, take the time to reread these examples carefully.

primality test	<pre>def isPrime(n):
	    if n < 2: return False
	    i = 2
	    while i*i <= n:
	        if n % i == 0: return False
	        i += 1
	    return True</pre>

hypotenuse of a right triangle	<pre>def hypot(a, b)
	    return math.sqrt(a*a + b*b)</pre>

generalized harmonic number	<pre>def harmonic(n, r=1):
	    total = 0.0
	    for i in range(1, n+1):
	        total += 1.0 / (i ** r)
	    return total</pre>

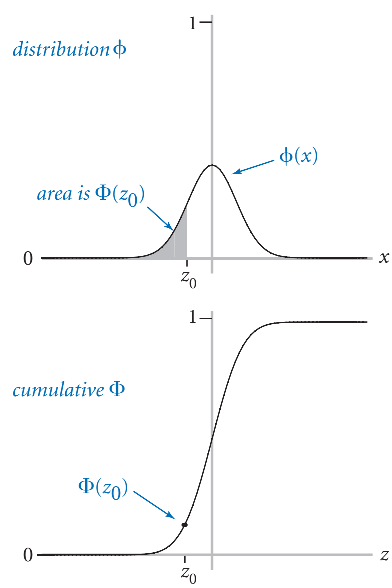
draw a triangle	<pre>def drawTriangle(x0, y0, x1, y1, x2, y2):
	    stddraw.line(x0, y0, x1, y1)
	    stddraw.line(x1, y1, x2, y2)
	    stddraw.line(x2, y2, x0, y0)</pre>

Typical code for implementing functions

Implementing mathematical functions Why not just use the Python built-in functions and those that are defined in the standard or extension Python modules? For example, why not use the `math.hypot()` function instead of defining our own `hypot()` function? The answer to this question is that we *do* use such functions when they are present (because they are likely to be faster and more accurate). However, there is an unlimited number of functions that we may wish to use and only a finite number of functions is defined in the Python standard and extension modules. When you need a function that is not defined in the Python standard or extension modules, you need to define the function yourself.

As an example, we consider the kind of code required for a familiar and important application that is of interest to many potential college students in the United States. In a recent year, over 1 million students took the Scholastic Aptitude Test (SAT). The test consists of two major sections: critical reading and mathematics. Scores range from 200 (lowest) to 800 (highest) on each section, so overall test scores range from 400 to 1600. Many universities consider these scores when making important decisions. For example, student athletes are required by the National Collegiate Athletic Association (NCAA), and thus by many universities, to have a combined score of at least 820 (out of 1600), and the minimum eligibility requirement for certain academic scholarships is 1500 (out of 1600). What percentage of test takers is ineligible for athletics? What percentage is eligible for the scholarships?

Two functions from statistics enable us to compute accurate answers to these questions. The *standard normal (Gaussian) probability density function* is characterized by the familiar bell-shaped curve and defined by the formula $\phi(x) = e^{-x^2/2}/\sqrt{2\pi}$. The standard normal (Gaussian) *cumulative distribution function* $\Phi(z)$ is defined to be the area under the curve defined by $\phi(x)$ above the x-axis and to the left of the vertical line $x=z$. These functions play an important role in science, engineering, and finance because they arise as accurate models throughout the natural world and because they are essential in understanding experimental error. In particular, these functions are known to accurately describe the distribution of test scores in our example, as a function of the mean (average value of the scores) and the standard deviation (square root of the average of the squares of the differences between each score and the mean), which are published each year. Given the mean μ and the standard deviation σ of the test scores, the percentage of students with scores less than a given value z is closely approximated by the function $\Phi(z, \mu, \sigma) = \Phi((z-\mu)/\sigma)$. Functions to calculate ϕ and Φ are not available in Python's `math` module, so we develop our own implementations.



Gaussian probability functions

Closed form. In the simplest situation, we have a closed-form mathematical formula defining our function in terms of functions that are implemented in Python's `math` module. This situation is the case for ϕ—the `math` module includes functions to compute the exponential and the square root functions (and a constant value for π), so a function `pdf()` corresponding to the mathematical definition is easy to implement. For convenience, `gauss.py` (PROGRAM 2.1.2) uses the default arguments $\mu = 0$ and $\sigma = 1$ and actually computes $\phi(x, \mu, \sigma) = \phi((x - \mu) / \sigma) / \sigma$.

No closed form. If no formula is known, we may need a more complicated algorithm to compute function values. This situation is the case for Φ—no closed-form expression exists for this function. Algorithms to compute function values sometimes follow immediately from Taylor series approximations, but developing reliably accurate implementations of mathematical functions is an art and a science that needs to be addressed carefully, taking advantage of the knowledge built up in mathematics over the past several centuries. Many different approaches have been studied for evaluating Φ. For example, a Taylor series approximation to the ratio of Φ and ϕ turns out to be an effective basis for evaluating the function:

$$\Phi(z) = 1/2 + \phi(z)\ (\ z + z^3/3 + z^5/(3{\cdot}5) + z^7/(3{\cdot}5{\cdot}7) + \dots\ ).$$

This formula readily translates to the Python code for the function `cdf()` in PROGRAM 2.1.2. For small (respectively large) z, the value is extremely close to 0 (respectively 1), so the code directly returns 0 (respectively 1); otherwise, it uses the Taylor series to add terms until the sum converges. Again, for convenience, PROGRAM 2.1.2 actually computes $\Phi(z, \mu, \sigma) = \Phi((z - \mu) / \sigma)$, using the defaults $\mu = 0$ and $\sigma = 1$.

Running `gauss.py` with the appropriate arguments on the command line tells us that about 17% of the test takers were ineligible for athletics in a year when the mean was 1019 and the standard deviation was 209. In the same year, about 1% percent qualified for academic scholarships.

COMPUTING WITH MATHEMATICAL FUNCTIONS OF ALL sorts plays a central role in science and engineering. In a great many applications, the functions that you need are expressed in terms of the functions in Python's `math` module, as we have just seen with `pdf()`, or in terms of a Taylor series approximation or some other formulation that is easy to compute, as we have just seen with `cdf()`. Indeed, support for such computations has played a central role throughout the evolution of computing systems and programming languages.

Program 2.1.2 Gaussian functions (gauss.py)

```
import math
import sys
import stdio

def pdf(x, mu=0.0, sigma=1.0):
    x = float(x - mu) / sigma
    return math.exp(-x*x/2.0) / math.sqrt(2.0*math.pi) / sigma

def cdf(z, mu=0.0, sigma=1.0):
    z = float(z - mu) / sigma
    if z < -8.0: return 0.0
    if z > +8.0: return 1.0
    total = 0.0
    term = z
    i = 3
    while total != total + term:
        total += term
        term *= z * z / i
        i += 2
    return 0.5 + total * pdf(z)

z     = float(sys.argv[1])
mu    = float(sys.argv[2])
sigma = float(sys.argv[3])
stdio.writeln(cdf(z, mu, sigma))
```

`total`	*cumulated sum*
`term`	*current term*

This code implements the Gaussian (normal) probability density (`pdf`) and cumulative distribution (`cdf`) functions, which are not implemented in Python's `math` library. The `pdf()` implementation follows directly from its definition, and the `cdf()` implementation uses a Taylor series and also calls `pdf()` (see accompanying text at left and Exercise 1.3.36). Note: If you are referring to this code for use in another program, please see `gaussian.py` (Program 2.2.1), which is designed for reuse.

```
% python gauss.py  820 1019 209
0.17050966869132106
% python gauss.py 1500 1019 209
0.9893164837383885
```

Using functions to organize code Beyond evaluating mathematical functions, the process of calculating an output value as a function of input values is important as a general technique for organizing control flow in *any* computation. Doing so is a simple example of an extremely important principle that is a prime guiding force for any good programmer: *Whenever you can clearly separate tasks within a computation, you should do so.*

Functions are natural and universal mechanism for expressing computational tasks. Indeed, the "bird's-eye view" of a Python program that we began with in Section 1.1 was equivalent to a function: we began by thinking of a Python program as a function that transforms command-line arguments into an output string. This view expresses itself at many different levels of computation. In particular, it is generally the case that you can express a long program more naturally in terms of functions instead of as a sequence of Python assignment, conditional, and loop statements. With the ability to define functions, you can better organize your programs by defining functions within them when appropriate.

For example, `coupon.py` (Program 2.1.3) on the facing page is an improved version of `couponcollector.py` (Program 1.4.2) that better separates the individual components of the computation. If you study Program 1.4.2, you will identify three separate tasks:

- Given the number of coupon values n, compute a random coupon value.
- Given n, do the coupon collection experiment.
- Get n from the command line, then compute and write the result.

Program 2.1.3 rearranges the code to reflect the reality that these three activities underlie the computation. The first two are implemented as functions, the third as global code.

With this organization, we could change `getCoupon()` (for example, we might want to draw the random numbers from a different distribution) or the global code (for example, we might want to take multiple inputs or run multiple experiments) without worrying about the effect of any of these changes on `collect()`.

Using functions isolates the implementation of each component of the collection experiment from others, or *encapsulates* them. Typically, programs have many independent components, which magnifies the benefits of separating them into different functions. We will discuss these benefits in further detail after we have seen several other examples, but you certainly can appreciate that it is better to express a computation in a program by breaking it up into functions, just as it is better to express an idea in an essay by breaking it up into paragraphs. *Whenever you can clearly separate tasks within a computation, you should do so.*

Program 2.1.3 Coupon collector (revisited) (coupon.py)

```
import random
import sys
import stdarray
import stdio

def getCoupon(n):
    return random.randrange(0, n)

def collect(n):
    isCollected = stdarray.create1D(n, False)
    count = 0
    collectedCount = 0
    while collectedCount < n:
        value = getCoupon(n)
        count += 1
        if not isCollected[value]:
            collectedCount += 1
            isCollected[value] = True
    return count

n = int(sys.argv[1])
result = collect(n)
stdio.writeln(result)
```

`n`	*# of coupon values (0 to n-1)*
`isCollected[i]`	*has coupon i been collected?*
`count`	*# of coupons collected*
`collectedCount`	*# of distinct coupons collected*
`value`	*value of current coupon*

This version of Program 1.4.2 illustrates the style of encapsulating computations in functions. This code has the same effect as `couponcollector.py`, but better separates the code into its three constituent pieces: generating a random integer between `0` and `n-1`, running a collection experiment, and managing the I/O.

```
% python coupon.py 1000
6522
% python coupon.py 1000
6481
% python coupon.py 1000000
12783771
```

Passing arguments and returning values Next, we examine the specifics of Python's mechanisms for passing arguments to and returning values from functions. These mechanisms are conceptually very simple, but it is worthwhile to take the time to understand them fully, as the effects are actually profound. Understanding argument-passing and return-value mechanisms is key to learning *any* new programming language. In the case of Python, the concepts of *immutability* and *aliasing* play a central role.

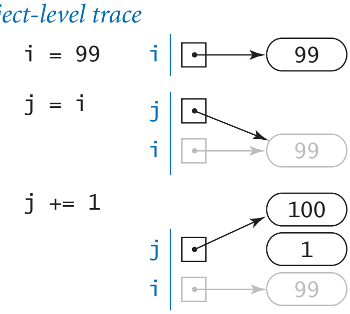
Call by object reference. You can use parameter variables anywhere in the body of the function in the same way as you use local variables. The only difference between a parameter variable and a local variable is that Python initializes the parameter variable with the corresponding argument provided by the calling code. We refer to this approach as *call by object reference*. (It is more commonly known as *call by value*, where the value is always an object reference—not the object's value.) One consequence of this approach is that if a parameter variable refers to a mutable object and you change that object's value within a function, then this also changes the object's value in the calling code (because it is the same object). Next, we explore the ramifications of this approach.

Immutability and aliasing. As discussed in Section 1.4, arrays are *mutable* data types, because we can change array elements. By contrast, a data type is *immutable* if it is not possible to change the value of an object of that type. The other data types that we have been using (`int`, `float`, `str`, and `bool`) are all immutable. In an immutable data type, operations that might seem to change a value actually result in the creation of a new object, as illustrated in the simple example at right. First, the statement `i = 99` creates an integer `99`, and assigns to `i` a reference to that integer. Then `j = i` assigns `i` (an object reference) to `j`, so both `i` and `j` reference the same object—the integer `99`. Two variables that reference the same objects are said to be *aliases*. Next, `j += 1` results in `j` referencing an object with value `100`, but *it does not do so by changing the value of the existing integer* from `99` to `100`! Indeed, since `int` objects are immutable, *no* statement

informal trace

	`i`	`j`
`i = 99`	99	
`j = i`	99	99
`j += 1`	99	100

object-level trace



Immutability of integers

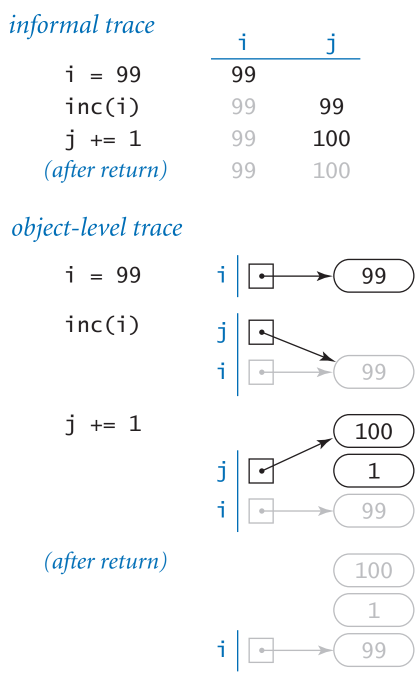
can change the value of that existing integer. Instead, that statement creates a new integer `1`, adds it to the integer `99` to create another new integer `100`, and assigns to `j` a reference to that integer. But `i` still references the original `99`. Note that the new integer `1` has no reference to it in the end—that is the system's concern, not ours. The immutability of integers, floats, strings, and booleans is a fundamental aspect of Python. We will consider the advantages and disadvantages of this approach in more detail in SECTION 3.3.

Integers, floats, booleans, and strings as arguments. The key point to remember about passing arguments to functions in Python is that *whenever you pass arguments to a function, the arguments and the function's parameter variables become aliases*. In practice, this is the predominant use of aliasing in Python, and it is important to understand its effects. For purposes of illustration, suppose that we need a function that increments an integer (our discussion applies to any more complicated function as well). A programmer new to Python might try this definition:

```
def inc(j):
    j += 1
```

and then expect to increment an integer `i` with the call `inc(i)`. Code like this would work in some programming languages, but it has no effect in Python, as shown in the figure at right. First, the statement `i = 99` assigns to global variable `i` a reference to the integer `99`. Then, the statement `inc(i)` passes `i`, an object reference, to the `inc()` function. That object reference is assigned to the parameter variable `j`. At this point `i` and `j` are aliases. As before, the `inc()` function's `j += 1` statement does not change the integer `99`, but rather creates a new integer `100` and assigns a reference to that integer to `j`. But when the `inc()` function returns to its caller, its parameter variable `j` goes out of scope, and the variable `i` still references the integer `99`.

This example illustrates that, in Python, *a function cannot produce the side effect of changing the value of an integer object* (nothing can do so). To increment variable `i`, we could use the definition



Aliasing in a function call

```
def inc(j):
    j += 1
    return j
```

and call the function with the assignment statement `i = inc(i)`.

The same holds true for any immutable type. A function cannot change the value of an integer, a float, a boolean, or a string.

Arrays as arguments. When a function takes an array as an argument, it implements a function that operates on an arbitrary number of objects. For example, the following function computes the mean (average) of an array of floats or integers:

```
def mean(a):
    total = 0.0
    for v in a:
        total += v
    return total / len(a)
```

We have been using arrays as arguments from the beginning of the book. For example, by convention, Python collects the strings that you type after the program name in the `python` command into an array `sys.argv[]` and implicitly calls your global code with that array of strings as the argument.

Side effects with arrays. Since arrays *are* mutable, it is often the case that the purpose of a function that takes an array as argument is to produce a side effect (such as changing the order of array elements). A prototypical example of such a function is one that exchanges the elements at two given indices in a given array. We can adapt the code that we examined at the beginning of SECTION 1.4:

```
def exchange(a, i, j):
   temp = a[i]
   a[i] = a[j]
   a[j] = temp
```

This implementation stems naturally from the Python array representation. The first parameter variable in `exchange()` is a reference to the array, not to all of the array's elements: when you pass an array as an argument to a function, you are giving it the opportunity to operate on that array (not a copy of it). A formal trace of a call on this function is shown on the facing page. This diagram is worthy of careful study to check your understanding of Python's function-call mechanism.

A second prototypical example of a function that takes an array argument and produces side effects is one that randomly shuffles the elements in the array, using this version of the algorithm that we examined in SECTION 1.4 (and the `exchange()` function just defined):

```
def shuffle(a):
    n = len(a)
    for i in range(n):
        r = random.randrange(i, n)
        exchange(a, i, r)
```

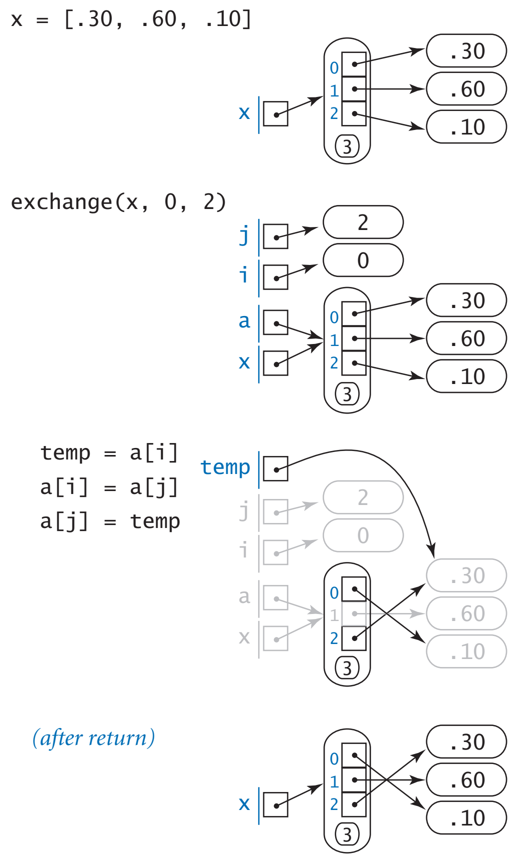
Incidentally, Python's standard function `random.shuffle()` does the same task. As another example, we will consider in SECTION 4.2 functions that sort an array (rearrange its elements so that they are in order).

Arrays as return values. A function that sorts, shuffles, or otherwise modifies an array taken as argument does not have to return a reference to that array, because it is changing the contents of a client array, not a copy. But there are many situations where it is useful for a function to provide an array as a return value. Chief among these are functions that create arrays for the purpose of returning multiple objects of the same type to a client.

As an example, consider the following function, which returns an array of random floats:

```
def randomarray(n):
    a = stdarray.create1D(n)
    for i in range(n):
        a[i] = random.random()
    return a
```

Later in this chapter, we will be developing numerous functions that return huge amounts of data in this way.



Exchanging two elements in an array

The table below concludes our discussion of arrays as function arguments by highlighting some typical array-procession functions.

mean of an array	<pre>def mean(a):
	    total = 0.0
	    for v in a:
	        total += v
	    return total / len(a)</pre>

dot product of two vectors of the same length	<pre>def dot(a, b):
	    total = 0
	    for i in range(len(a)):
	        total += a[i] * b[i]
	    return total</pre>

exchange two elements in an array	<pre>def exchange(a, i, j):
	    temp = a[i]
	    a[i] = a[j]
	    a[j] = temp</pre>

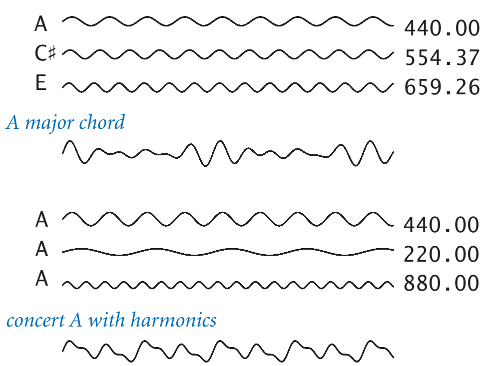
write a one-dimensional array (and its length)	<pre>def write1D(a):
	    stdio.writeln(len(a))
	    for v in a:
	        stdio.writeln(v)</pre>

read a two-dimensional array of floats (with dimensions)	<pre>def readFloat2D():
	    m = stdio.readInt()
	    n = stdio.readInt()
	    a = stdarray.create2D(m, n, 0.0)
	    for i in range(m):
	        for j in range(n):
	            a[i][j] = stdio.readFloat()
	    return a</pre>

Typical code for implementing functions with arrays

Example: superposition of sound waves As discussed in SECTION 1.5, the simple audio model that we studied there needs to be embellished to create sound that resembles the sound produced by a musical instrument. Many different embellishments are possible; with functions, we can systematically apply them to produce sound waves that are far more complicated than the simple sine waves that we produced in SECTION 1.5. As an illustration of the effective use of functions to solve an interesting computational problem, we consider a program that has essentially the same functionality as `playthattune.py` (PROGRAM 1.5.8), but adds harmonic tones one octave above and one octave below each note to produce a more realistic sound.

Chords and harmonics. Notes like concert *A* have a pure sound that is not very musical, because the sounds that you are accustomed to hearing have many other components. The sound from a guitar string echoes off the wooden part of the instrument, the walls of the room that you are in, and so forth. You may think of such effects as modifying the basic sine wave. For example, most musical instruments produce *harmonics* (the same note in different octaves and not as loud), or you might play *chords* (multiple notes at the same time). To combine multiple sounds, we use *superposition*: simply add their waves together and rescale to make sure that all values stay between -1 and $+1$. As it turns out, when we superpose sine waves of different frequencies in this way, we can get arbitrarily complicated waves. Indeed, one of the triumphs of 19th-century mathematics was the development of the idea that any smooth periodic function can be expressed as a sum of sine and cosine waves, known as a *Fourier series*. This mathematical idea corresponds to the notion that we can create a large range of sounds with musical instruments or our vocal cords and that all sound consists of a composition of various oscillating curves. Any sound corresponds to a curve and any curve corresponds to a sound, so we can create arbitrarily complex curves with superposition.



Superposing waves to make composite sounds

Computing with sound waves. In Section 1.5, we saw how to represent sound waves by arrays of numbers that represent their values at the same sample points. Now, we will use such arrays as return values and arguments to functions to process such data. For example, the following function takes a frequency (in hertz) and a duration (in seconds) as arguments and returns a representation of a sound wave (more precisely, an array that contains values sampled from the specified wave at the standard 44,100 samples per second).

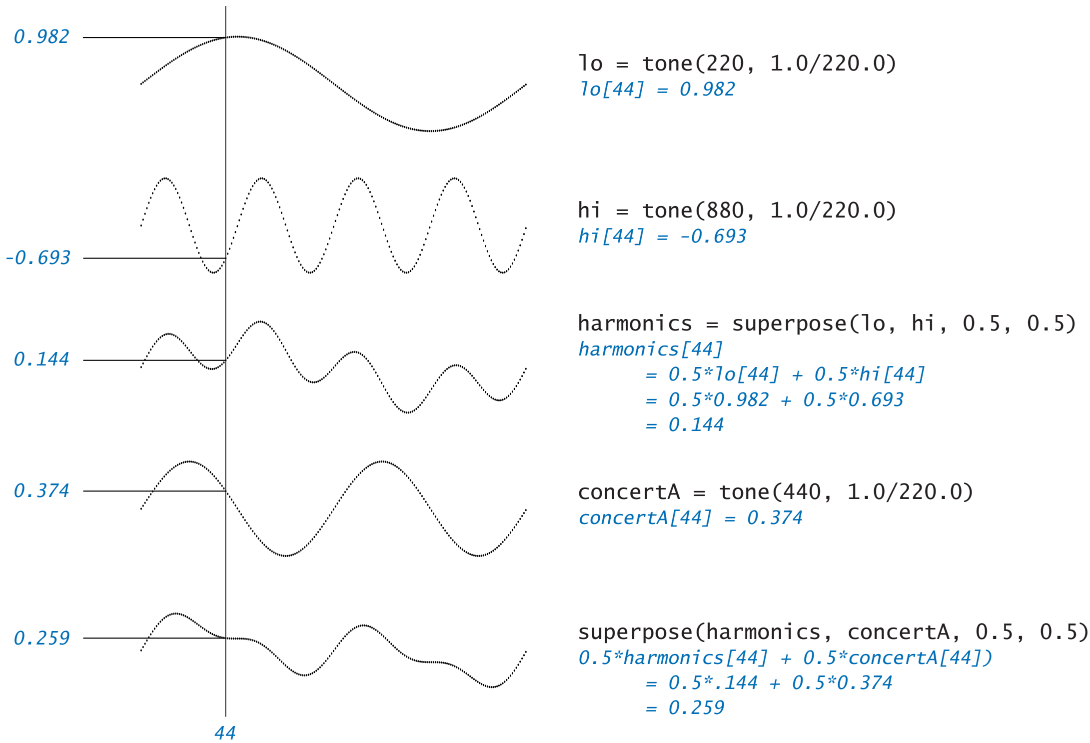
```
def tone(hz, duration, sps=44100):
    n = int(sps * duration)
    a = stdarray.create1D(n+1, 0.0)
    for i in range(n+1):
        a[i] = math.sin(2.0 * math.pi * i * hz / sps)
    return a
```

The size of the array returned depends on the duration: it contains about `sps*duration` floats (nearly half a million floats for 10 seconds). But we can now treat that array (the value returned from tone) as a single entity and compose code that processes sound waves, as we will soon see in Program 2.1.4.

Weighted superposition. Since we represent sound waves by arrays of numbers that represent their values at the same sample points, superposition is simple to implement: we add together their sample values at each sample point to produce the combined result. For greater control, we also specify a relative weight for each of the two waves to be superposed, with the following function:

```
def superpose(a, b, aWeight, bWeight):
    c = stdarray.create1D(len(a), 0.0)
    for i in range(len(a)):
        c[i] = aWeight*a[i] + bWeight*b[i]
    return c
```

(This code assumes that `a[]` and `b[]` are of the same length.) For example, if we have a sound represented by an array `a[]` that we want to have three times the effect of the sound represented by an array `b[]`, we would call `superpose(a, b, 0.75, 0.25)`. The figure at the top of the next page shows the use of two calls on this function to add harmonics to a tone (we superpose the harmonics, then superpose the result with the original tone, which has the effect of giving the original tone twice



Adding harmonics to concert A (1/220 second at 44,100 samples/second)

the weight of each harmonic). As long as the weights are positive and sum to 1, `superpose()` preserves our convention of keeping the values of all waves between −1 and +1.

Program 2.1.4 (`playthattunedeluxe.py`) is an implementation that applies these concepts to produce a more realistic sound than that produced by Program 1.5.8. To do so, it makes use of functions to divide the computation into four parts:

- Given a frequency and duration, create a pure tone.
- Given two sound waves and relative weights, superpose them.
- Given a pitch and duration, create a note with harmonics.
- Read and play a sequence of pitch/duration pairs from standard input.

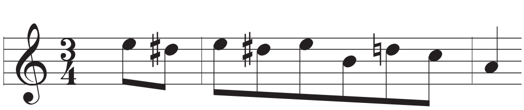
Program 2.1.4 Play that tune (revisited) (`playthattunedeluxe.py`)

```
import math
import stdarray
import stdaudio
import stdio

def superpose(a, b, aWeight, bWeight):
    c = stdarray.create1D(len(a), 0.0)
    for i in range(len(a)):
        c[i] = aWeight*a[i] + bWeight*b[i]
    return c

def tone(hz, duration, sps=44100):
    n = int(sps * duration)
    a = stdarray.create1D(n+1, 0.0)
    for i in range(n+1):
        a[i] = math.sin(2.0 * math.pi * i * hz / sps)
    return a

def note(pitch, duration):
    hz = 440.0 * (2.0 ** (pitch / 12.0))
    lo = tone(hz/2, duration)
    hi = tone(2*hz, duration)
    harmonics = superpose(lo, hi, 0.5, 0.5)
    a = tone(hz, duration)
    return superpose(harmonics, a, 0.5, 0.5)

while not stdio.isEmpty():
    pitch = stdio.readInt()
    duration = stdio.readFloat()
    a = note(pitch, duration)
    stdaudio.playSamples(a)
stdaudio.wait()
```

`hz`	*frequency*
`lo[]`	*lower harmonic*
`hi[]`	*upper harmonic*
`h[]`	*combined harmonics*
`a[]`	*pure tone*

This program reads sound samples, embellishes the sounds by adding harmonics to create a more realistic tone than Program 1.5.8, and plays the resulting sound to standard audio.

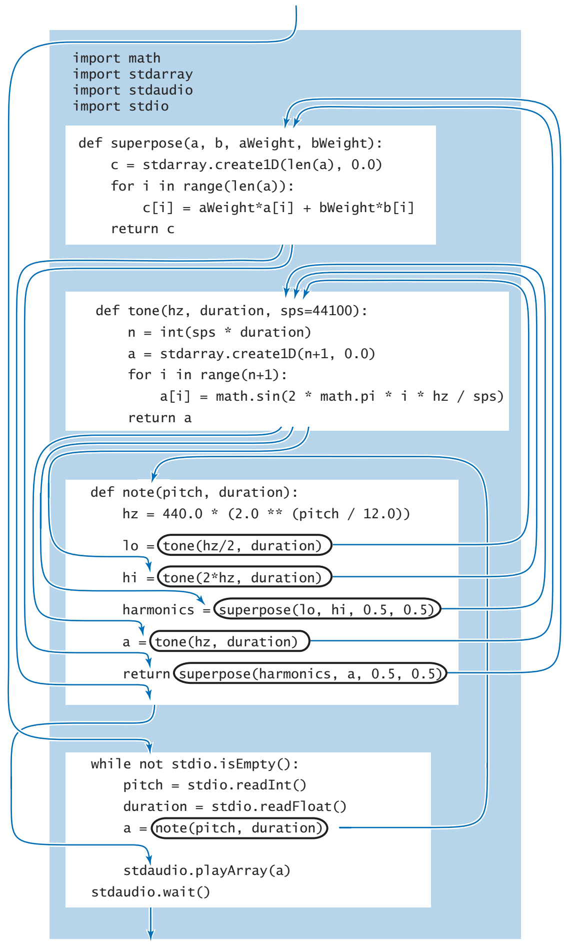
```
% python playthattunedeluxe.py < elise.txt
```



```
% more elise.txt
7 .125  6 .125
7 .125  6 .125  7 .125
2 .125  5 .125  3 .125
0 .25
```

These tasks are all amenable to implementation as functions, which depend on one another. Each function is well defined and straightforward to implement. All of them (and `stdaudio`) represent sound as a series of discrete values kept in an array, corresponding to sampling a sound wave at 44,100 samples per second.

Up to this point, our use of functions has been somewhat of a notational convenience. For example, the control flow in Program 2.1.1, Program 2.1.2, and Program 2.1.3 is simple—each function is called in just one place in the code. By contrast, Program 2.1.4 is a convincing example of the effectiveness of defining functions to organize a computation because each function is called multiple times. For example, as illustrated in the figure below, the function `note()` calls the function `tone()` three times and the function `superpose()` twice. Without functions, we would need multiple copies of the code in `tone()` and `superpose()`; with functions, we can deal directly with concepts close to the application. Like loops, functions have a simple but profound effect: one sequence of statements (those in the function definition) is executed multiple times during the execution of our program—once for each time the function is called in the control flow in the global code.



Flow of control among several functions

Functions are important because they give us the ability to *extend* the Python language within a program. Having implemented and debugged functions such as `harmonic()`, `pdf()`, `cdf()`, `mean()`, `exchange()`, `shuffle()`, `isPrime()`, `superpose()`, `tone()`, and `note()`, we can use them almost as if they were built into Python. The flexibility to do so opens up a whole new world of programming. Before, you were safe in thinking about a Python program as a sequence of statements. Now you need to think of a Python program as a *set of functions* that can call one another. The statement-to-statement control flow to which you have been accustomed is still present within functions, but programs have a higher-level control flow defined by function calls and returns. This ability enables you to think in terms of operations called for by the application, not just the operations that are built into Python.

Whenever you can clearly separate tasks within a computation, you should do so. The examples in this section (and the programs throughout the rest of the book) clearly illustrate the benefits of adhering to this maxim. With functions, we can

- Divide a long sequence of statements into independent parts.
- Reuse code without having to copy it.
- Work with higher-level concepts (such as sound waves).

This point of view leads to code that is easier to understand, maintain, and debug compared to a long program composed solely of Python assignment, conditional, and loop statements. In the next section, we discuss the idea of using functions defined in other files, which again takes us to another level of programming.

Q&A

Q. Can I use the statement `return` in a function without specifying a value?

A. Yes. Technically, it returns the `None` object, which is the sole value of the type `NoneType`.

Q. What happens if a function has one control flow that leads to a `return` statement that returns a value but another control flow that reaches the end of the function body?

A. It would be poor style to define such a function, because doing so would place a severe burden on the function's callers: the callers would need to know under which circumstances the function returns a value, and under which circumstances it returns `None`.

Q. What happens if I compose code in the body of a function that appears after the `return` statement?

A. Once a `return` statement is reached, control returns to the caller. So any code in the body of a function that appears after a `return` statement is useless; it is never executed. In Python, it is poor style, but not illegal to define such a function.

Q. What happens if I define two functions with the same name (but possibly a different number of arguments) in the same `.py` file?

A. This is known as *function overloading*, which is embraced by many programming languages. Python, however, is not one of those languages: the second function definition will overwrite the first one. You can often achieve the same effect by using default arguments.

Q. What happens if I define two functions with the same name in different files?

A. That is fine. For example, it would be good design to have a function named `pdf()` in `gauss.py` that computes the Gaussian probability density function and another function named `pdf()` in `cauchy.py` that computes the Cauchy probability density function. In SECTION 2.2 you will learn how to call functions defined in different `.py` files.

Q. Can a function change the object to which a parameter variable is bound?

A. Yes, you can use a parameter variable on the left side of an assignment statement. However, many Python programmers consider it poor style to do so. Note that such an assignment statement has no effect in the client.

Q. The issue with side effects and mutable objects is complicated. Is it really all that important?

A. Yes. Properly controlling side effects is one of a programmer's most important tasks in large systems. Taking the time to be sure that you understand the difference between passing arrays (which are mutable) and passing integers, floats, booleans, and strings (which are immutable) will certainly be worthwhile. The very same mechanisms are used for all other types of data, as you will learn in Chapter 3.

Q. How can I arrange to pass an array to a function in such a way that the function cannot change the elements in the array?

A. There is no direct way to do so. In Section 3.3 you will see how to achieve the same effect by building a *wrapper* data type and passing an object of that type instead. You will also see how to use Python's built-in `tuple` data type, which represents an immutable sequence of objects.

Q. Can I use a mutable object as a default value for an optional argument?

A. Yes, but it may lead to unexpected behavior. Python evaluates a default value only once, when the function is defined (not each time the function is called). So, if the body of a function modifies a default value, subsequent function calls will use the modified value. Similar difficulties arise if you initialize the default value by calling an impure function. For example, after Python executes the code fragment

```
def append(a=[], x=random.random()):
    a += [x]
    return a

b = append()
c = append()
```

`b[]` and `c[]` are aliases for the same array of length 2 (not 1), which contains one float repeated twice (instead of two different floats).

Exercises

2.1.1 Compose a function `max3()` that takes three `int` or `float` arguments and returns the largest one.

2.1.2 Compose a function `odd()` that takes three `bool` arguments and returns `True` if an odd number of arguments are `True`, and `False` otherwise.

2.1.3 Compose a function `majority()` that takes three `bool` arguments and returns `True` if at least two of the arguments are `True`, and `False` otherwise. Do not use an `if` statement.

2.1.4 Compose a function `areTriangular()` that takes three numbers as arguments and returns `True` if they could be lengths of the sides of a triangle (none of them is greater than or equal to the sum of the other two), and `False` otherwise.

2.1.5 Compose a function `sigmoid()` that takes a `float` argument `x` and returns the `float` obtained from the formula $1 / (1+e^{-x})$.

2.1.6 Compose a function `lg()` that takes an integer `n` as an argument and returns the base-2 logarithm of `n`. You may use Python's `math` module.

2.1.7 Compose a function `lg()` that takes an integer `n` as an argument and returns the largest integer not larger than the base-2 logarithm of `n`. Do *not* use the `math` module.

2.1.8 Compose a function `signum()` that takes a `float` argument `n` and returns `-1` if `n` is less than `0`, `0` if `n` is equal to `0`, and `+1` if `n` is greater than `0`.

2.1.9 Consider this function `duplicate()`:

```
def duplicate(s):
    t = s + s
```

What does the following code fragment write?

```
s = 'Hello'
s = duplicate(s)
t = 'Bye'
t = duplicate(duplicate(duplicate(t)))
stdio.writeln(s + t)
```

2.1.10 Consider this function `cube()`:

```
def cube(i):
   i = i * i * i
```

How many times is the following `while` loop iterated?

```
i = 0
while i < 1000:
   cube(i)
   i += 1
```

Solution: Just 1,000 times. A call to `cube()` has no effect on the client code. It changes the parameter variable `i`, but that change has no effect on the variable `i` in the `while` loop, which is a different variable. If you replace the call to `cube(i)` with the statement `i = i * i * i` (maybe that was what you were thinking), then the loop is iterated five times, with `i` taking on the values `0`, `1`, `2`, `9`, and `730` at the beginning of the five iterations.

2.1.11 What does the following code fragment write?

```
for i in range(5):
    stdio.write(i)
for j in range(5):
    stdio.write(i)
```

Solution: `0123444444`. Note that the second call to `stdio.write()` uses `i`, not `j`. Unlike analogous loops in many other programming languages, when the first `for` loop terminates, the variable `i` is `4` and it remains in scope.

2.1.12 The following *checksum* formula is widely used by banks and credit card companies to validate legal account numbers:

$$d_0 + f(d_1) + d_2 + f(d_3) + d_4 + f(d_5) + \ldots = 0 \pmod{10}$$

The d_i are the decimal digits of the account number and $f(d)$ is the sum of the decimal digits of $2d$ (for example, $f(7) = 5$ because $2 \times 7 = 14$ and $1 + 4 = 5$). For example 17327 is valid because $1 + 5 + 3 + 4 + 7 = 20$, which is a multiple of 10.

Implement the function *f* and compose a program to take a 10-digit integer as a command-line argument and write a valid 11-digit number with the given integer as its first 10 digits and the checksum as the last digit.

2.1.13 Given two stars with angles of declination and right ascension (d_1, a_1) and (d_2, a_2), respectively, the angle they subtend is given by the formula

$$2 \arcsin((\sin^2(d/2) + \cos(d_1)\cos(d_2)\sin^2(a/2))^{1/2}),$$

where a_1 and a_2 are angles between -180 and 180 degrees, d_1 and d_2 are angles between -90 and 90 degrees, $a = a_2 - a_1$, and $d = d_2 - d_1$. Compose a program to take the declination and right ascension of two stars as command-line arguments and write the angle they subtend. *Hint*: Be careful about converting from degrees to radians.

2.1.14 Compose a `readBool2D()` function that reads a two-dimensional matrix of 0 and 1 values (with dimensions) into an array of booleans.

Solution: The body of the function is virtually the same as for the corresponding function given in the table in the text for two-dimensional arrays of floats:

```
def readBool2D():
    m = stdio.readInt()
    n = stdio.readInt()
    a = stdarray.create2D(m, n, False)
    for i in range(m):
        for j in range(n):
            a[i][j] = stdio.readBool()
    return a
```

2.1.15 Compose a function that takes an array `a[]` of strictly positive floats as its argument and rescales the array so that each element is between 0 and 1 (by subtracting the minimum value from each element and then dividing each element by the difference between the minimum and maximum values). Use the built-in `max()` and `min()` functions.

2.1.16 Compose a function `histogram()` that takes an array `a[]` of integers and an integer `m` as arguments and returns an array of length `m` whose `i`th element is the number of times the integer `i` appears in the argument array. Assume that the values in `a[]` are all between `0` and `m-1`, so that the sum of the values in the returned array should be equal to `len(a)`.

2.1.17 Assemble code fragments in this section and in SECTION 1.4 to develop a program that takes an integer `n` from the command line and writes `n` five-card hands, separated by blank lines, drawn from a randomly shuffled card deck, one card per line using card names like `Ace of Clubs`.

2.1.18 Compose a function `multiply()` that takes two square matrices of the same dimension as arguments and returns their product (another square matrix of that same dimension). *Extra credit*: Make your program work whenever the number of columns in the first matrix is equal to the number of rows in the second matrix.

2.1.19 Compose a function `any()` that takes an array of booleans as an argument and returns `True` if *any* of the elements in the array is `True`, and `False` otherwise. Compose a function `all()` that takes an array of booleans as an argument and returns `True` if *all* of the elements in the array are `True`, and `False` otherwise. Note that `all()` and `any()` are built-in Python functions; the goal of this exercise is to understand them better by creating your own versions.

2.1.20 Develop a version of `getCoupon()` that better models the situation when one of the n coupons is rare: choose one value at random, return that value with probability $1/(1000n)$, and return all other values with equal probability. *Extra credit*: How does this change affect the average value of the coupon collector function?

2.1.21 Modify `playthattune.py` to add harmonics two octaves away from each note, with half the weight of the one-octave harmonics.

Creative Exercises

2.1.22 *Birthday problem.* Compose a program with appropriate functions for studying the birthday problem (see Exercise 1.4.35).

2.1.23 *Euler's totient function.* Euler's totient function is an important function in number theory: $\varphi(n)$ is defined as the number of positive integers less than or equal to n that are relatively prime with n (no factors in common with n other than 1). Compose a function that takes an integer argument n and returns $\varphi(n)$. Include global code that takes an integer from the command line, calls the function, and writes the result.

2.1.24 *Harmonic numbers.* Create a program `harmonic.py` that defines three functions `harmonic()`, `harmonicSmall()`, and `harmonicLarge()` for computing the harmonic numbers. The `harmonicSmall()` function should just compute the sum (as in Program 2.1.1), the `harmonicLarge()` function should use the approximation $H_n = \log_e(n) + \gamma + 1/(2n) - 1/(12n^2) + 1/(120n^4)$ (the number $\gamma = 0.577215664901532\ldots$ is known as *Euler's constant*), and the `harmonic()` function should call `harmonicSmall()` for $n < 100$ and `harmonicLarge()` otherwise.

2.1.25 *Gaussian random values.* Experiment with the following function for generating random variables from the Gaussian distribution, which is based on generating a random point in the unit circle and using a form of the Box-Muller formula (see Exercise 1.2.24).

```
def gaussian():
    r = 0.0
    while (r >= 1.0) or (r == 0.0):
        x = -1.0  + 2.0 * random.random()
        y = -1.0  + 2.0 * random.random()
        r = x*x + y*y
    return x * math.sqrt(-2.0 * math.log(r) / r)
```

Take a command-line argument `n` and generate `n` random numbers, using an array `a[]` of 20 integers to count the numbers generated that fall between `i*.05` and `(i+1)*.05` for `i` from `0` to `19`. Then use `stddraw` to plot the values and to compare your result with the normal bell curve. *Remark*: This approach is faster and more accurate than the one described in Exercise 1.2.24. Although it involves a loop, the loop is executed only 4 / π (about 1.273) times on average. This reduces the overall expected number of calls to transcendental functions.

2.1.26 *Binary search.* A general function that we study in detail in Section 4.2 is effective for computing the inverse of a cumulative distribution function like `cdf()`. Such functions are continuous and nondecreasing from (0,0) to (1,1). To find the value x_0 for which $f(x_0) = y_0$, check the value of $f(0.5)$. If it is greater than y_0, then x_0 must be between 0 and 0.5; otherwise, it must be between 0.5 and 1. Either way, we halve the length of the interval known to contain x_0. Iterating, we can compute x_0 to within a given tolerance. Add a function `cdfInverse()` to `gauss.py` that uses binary search to compute the inverse. Change the global code to take a number p between 0 and 100 as a third command-line argument and write the minimum score that a student would need to be in the top p percent of students taking the SAT in a year when the mean and standard deviation were the first two command-line arguments.

2.1.27 *Black-Scholes option valuation.* The Black-Scholes formula supplies the theoretical value of a European call option on a stock that pays no dividends, given the current stock price s, the exercise price x, the continuously compounded risk-free interest rate r, the standard deviation σ of the stock's return (volatility), and the time (in years) to maturity t. The value is given by the formula $s\,\Phi(a) - x\,e^{-rt}\,\Phi(b)$, where $\Phi(z)$ is the Gaussian cumulative distribution function, $a = (\ln(s/x) + (r + \sigma^2/2)\,t) / (\sigma\sqrt{t})$, and $b = a - \sigma\sqrt{t}$. Compose a program that takes `s`, `x`, `r`, `sigma`, and `t` from the command line and writes the Black-Scholes value.

2.1.28 *Implied volatility.* Typically the volatility is the unknown value in the Black-Scholes formula. Compose a program that reads `s`, `x`, `r`, `t`, and the current price of the European call option from the command line and uses binary search (see Exercise 2.1.26) to compute σ.

2.1.29 *Horner's method.* Compose a program `horner.py` with a function `evaluate(x, a)` that evaluates the polynomial $a(x)$ whose coefficients are the elements in the array `a[]`:

$$a_0 + a_1x^1 + a_2x^2 + \ldots + a_{n-2}x^{n-2} + a_{n-1}x^{n-1}$$

Use *Horner's method*, an efficient way to perform the computations that is suggested by the following parenthesization:

$$a_0 + x\,(a_1 + x\,(a_2 + \ldots + x\,(a_{n-2} + x a_{n-1})\ldots))$$

Then compose a function `exp()` that calls `evaluate()` to compute an approximation to e^x, using the first n terms of the Taylor series expansion $e^x = 1 + x + x^2/2! + x^3/3! + \ldots$. Take an argument `x` from the command line, and compare your result against that computed by `math.exp(x)`.

2.1.30 *Benford's law.* The American astronomer Simon Newcomb observed a quirk in a book that compiled logarithm tables: the beginning pages were much grubbier than the ending pages. He suspected that scientists performed more computations with numbers starting with 1 than with 8 or 9, and postulated the first digit law, which says that under general circumstances, the leading digit is much more likely to be 1 (roughly 30%) than 9 (less than 4%). This phenomenon is known as *Benford's law* and is now often used as a statistical test. For example, IRS forensic accountants rely on it to discover tax fraud. Compose a program that reads in a sequence of integers from standard input and tabulates the number of times each of the digits 1–9 is the leading digit, breaking the computation into a set of appropriate functions. Use your program to test the law on some tables of information from your computer or from the web. Then, compose a program to foil the IRS by generating random amounts from \$1.00 to \$1,000.00 with the same distribution that you observed.

2.1.31 *Binomial distribution.* Compose a function `binomial()` that accepts an integer `n`, an integer `k`, and a float `p`, and computes the probability of obtaining exactly `k` heads in `n` biased coin flips (heads with probability `p`) using the formula

$$f(k, n, p) = p^k (1-p)^{n-k}\, n!\,/\,(k!(n-k)!)$$

Hint: To avoid computing with huge integers, compute $x = \ln f(k, n, p)$ and then return e^x. In the global code, take `n` and `p` from the command line and check that the sum over all values of `k` between 0 and `n` is (approximately) 1. Also, compare every value computed with the normal approximation

$$f(k, n, p) \;\approx\; \Phi(k + 1/2, np, \sqrt{np(1-p)}) \;-\; \Phi(k - 1/2, np, \sqrt{np(1-p)})$$

2.1.32 *Coupon collecting from a binomial distribution.* Compose a version of `getCoupon()` that uses `binomial()` from the previous exercise to return coupon values according to the binomial distribution with $p = 1/2$. *Hint*: Generate a uniformly distributed random number x between 0 and 1, then return the smallest value of k for which the sum of $f(j, n, p)$ for all $j < k$ exceeds x. *Extra credit*: Develop a hypothesis for describing the behavior of the coupon collector function under this assumption.

2.1.33 *Chords.* Compose a version of `playthattunedeluxe.py` that can handle songs with chords (three or more different notes, including harmonics). Develop an input format that allows you to specify different durations for each chord and different amplitude weights for each note within a chord. Create test files that exercise your program with various chords and harmonics, and create a version of *Für Elise* that uses them.

2.1.34 *Postal barcodes.* The barcode used by the U.S. Postal System to route mail is defined as follows: Each decimal digit in the ZIP code is encoded using a sequence of three half-height and two full-height bars. The barcode starts and ends with a full-height bar (the guard rail) and includes a checksum digit (after the five-digit ZIP code or ZIP+4), computed by summing up the original digits modulo 10. Define the following functions:

- Draw a half-height or full-height bar on `stddraw`.
- Given a digit, draw its sequence of bars.
- Compute the checksum digit.

Also define global code that reads in a five- (or nine-) digit ZIP code as the command-line argument and draws the corresponding postal barcode.

2.1.35 *Calendar.* Compose a program `cal.py` that takes two command-line arguments `m` and `y` and writes the monthly calendar for the `m`th month of year `y`, as in this example:

```
% python cal.py 2 2015
February 2015
 S  M Tu  W Th  F  S
 1  2  3  4  5  6  7
 8  9 10 11 12 13 14
15 16 17 18 19 20 21
22 23 24 25 26 27 28
```

Hint: See `leapyear.py` (Program 1.2.5) and Exercise 1.2.26.

2.1.36 *Fourier spikes.* Compose a program that takes a command-line argument n and plots the function

$$(\cos(t) + \cos(2t) + \cos(3t) + \ldots + \cos(Nt)) \,/\, N$$

for 500 equally spaced samples of t from -10 to 10 (in radians). Run your program for $n = 5$ and $n = 500$. *Note*: You will observe that the sum converges to a spike (0 everywhere except a single value). This property is the basis for a proof that *any* smooth function can be expressed as a sum of sinusoids.

This page intentionally left blank

Index

F

S