

Programming and Mathematical Thinking

**A Gentle Introduction to Discrete Math
Featuring Python**

Allan M. Stavelly

The New Mexico Tech Press
Socorro, New Mexico, USA

Programming and Mathematical Thinking

A Gentle Introduction to Discrete Math Featuring Python

Allan M. Stavely

Copyright © 2014 Allan M. Stavely

First Edition

Content of this book available under the Creative Commons Attribution-Noncommercial-ShareAlike License. See <http://creativecommons.org/licenses/by-nc-sa/4.0/> for details.

Publisher's Cataloguing-in-Publication Data

Stavely, Allan M

Programming and mathematical thinking: a gentle introduction to discrete math featuring Python / Allan M. Stavely.

xii, 246 p.: ill. ; 28 cm

ISBN 978-1-938159-00-8 (pbk.) — 978-1-938159-01-5 (ebook)

1. Computer science — Mathematics. 2. Mathematics — Discrete Mathematics. 3. Python (Computer program language).

QA 76.9 .M35 .S79 2014
004-dc22

OCLC Number: 863653804

Published by The New Mexico Tech Press, a New Mexico nonprofit corporation

The New Mexico Tech Press
Socorro, New Mexico, USA
<http://press.nmt.edu>

Once more, to my parents, Earl and Ann

Table of Contents

Preface	vii
1. Introduction	1
1.1. Programs, data, and mathematical objects	1
1.2. A first look at Python	3
1.3. A little mathematical terminology	10
2. An overview of Python	17
2.1. Introduction	17
2.2. Values, types, and names	18
2.3. Integers	19
2.4. Floating-point numbers	23
2.5. Strings	25
3. Python programs	29
3.1. Statements	29
3.2. Conditionals	31
3.3. Iterations	35
4. Python functions	41
4.1. Function definitions	41
4.2. Recursive functions	43
4.3. Functions as values	45
4.4. Lambda expressions	48
5. Tuples	51
5.1. Ordered pairs and n -tuples	51
5.2. Tuples in Python	52
5.3. Files and databases	54
6. Sequences	57
6.1. Properties of sequences	57
6.2. Monoids	59
6.3. Sequences in Python	64
6.4. Higher-order sequence functions	67
6.5. Comprehensions	73
6.6. Parallel processing	74
7. Streams	83
7.1. Dynamically-generated sequences	83
7.2. Generator functions	85

7.3. Endless streams	90
7.4. Concatenation of streams	92
7.5. Programming with streams	95
7.6. Distributed processing	103
8. Sets	107
8.1. Mathematical sets	107
8.2. Sets in Python	110
8.3. A case study: finding students for jobs	114
8.4. Flat files, sets, and tuples	118
8.5. Other representations of sets	123
9. Mappings	127
9.1. Mathematical mappings	127
9.2. Python dictionaries	131
9.3. A case study: finding given words in a file of text	135
9.4. Dictionary or function?	140
9.5. Multisets	145
10. Relations	153
10.1. Mathematical terminology and notation	153
10.2. Representations in programs	156
10.3. Graphs	159
10.4. Paths and transitive closure	164
10.5. Relational database operations	167
11. Objects	175
11.1. Objects in programs	175
11.2. Defining classes	177
11.3. Inheritance and the hierarchy of classes	180
11.4. Object-oriented programming	184
11.5. A case study: moving averages	185
11.6. Recursively-defined objects: trees	194
11.7. State machines	201
12. Larger programs	213
12.1. Sharing tune lists	213
12.2. Biological surveys	218
12.3. Note cards for writers	227
Afterword	233
Solutions to selected exercises	235
Index	241

List of Examples

1.1. Finding a name	4
1.2. Finding an email address	7
1.3. Average of a collection of observations	8
6.1. Finding a name again, in functional style	71
6.2. Average of observations again, in functional style	72
7.1. Combinations using a generator function	89
8.1. Finding job candidates using set operations	117
8.2. Job candidates again, with different input files	122
9.1. Finding given words in a document	139
9.2. A memoized function: the n^{th} Fibonacci number	144
9.3. Number of students in each major field	149
10.1. Distances using symmetry and reflexivity	159
11.1. The <code>MovingAverage</code> class	191
11.2. The <code>MovingAverage</code> class, version 2	193
11.3. The <code>PushButton</code> class	204
11.4. A state machine for finding fields in a string	206
11.5. Code that uses a <code>FieldsStateMachine</code>	207
11.6. A state machine for finding fields, version 2	209

Preface

My mission in this book is to encourage programmers to think mathematically as they develop programs.

This idea is nothing new to programmers in science and engineering fields, because much of their work is inherently based on numerical mathematics and the mathematics of real numbers. However, there is more to mathematics than numbers.

Some of the mathematics that is most relevant to programming is known as “discrete mathematics”. This is the mathematics of discrete elements, such as symbols, character strings, truth values, and “objects” (to use a programming term) that are collections of properties. Discrete mathematics is concerned with such elements; collections of them, such as sets and sequences; and connections among elements, in structures such as mappings and relations. In many ways discrete mathematics is more relevant to programming than numerical mathematics is: not just to particular kinds of programming, but to all programming.

Many experienced programmers approach the design of a program by describing its input, output, and internal data objects in the vocabulary of discrete mathematics: sets, sequences, mappings, relations, and so on. This is a useful habit for us, as programmers, to cultivate. It can help to clarify our thinking about design problems; in fact, solutions often become obvious. And we inherit a well-understood vocabulary for specifying and documenting our programs and for discussing them with other programmers.¹

For example, consider this simple programming problem. Suppose that we are writing software to analyze web pages, and we want some code that will read two web pages and find all of the URLs that appear in both. Some programmers might approach the problem like this:

¹This paragraph and the example that follows are adapted from a previous book: Allan M. Staveland, *Toward Zero-Defect Programming* (Reading, Mass.: Addison Wesley Longman, 1999), 142–143.

First I'll read the first web page and store all the URLs I find in a list. Then I'll read the second web page and, every time I find a URL, search the list for it. But wait: I don't want to include the same URL in my result more than once. I'll keep a second list of the URLs that I've already found in both web pages, and search that before I search the list of URLs from the first web page.

But a programmer who is accustomed to thinking in terms of discrete-mathematical structures might immediately think of a different approach:

The URLs in a web page are a set. I'll read each web page and build up the set of URLs in each using set insertion. Then I can get the URLs common to both web pages by using set intersection.

Either approach will work, but the second is conceptually simpler, and it will probably be more straightforward to implement. In fact, once the problem is described in mathematical terms, most of the design work is already done.

That's the kind of thinking that this book promotes.

As a vehicle, I use the programming language Python. It's a clean, modern language, and it comes with many of the mathematical structures that we will need: strings, sets, several kinds of sequences, finite mappings (dictionaries, which are more general than arrays), and functions that are first-class values. All these are built into the core of the language, not add-ons implemented by libraries as in many programming languages. Python is easy to get started with and makes a good first language, far better than C or C++ or Java, in my opinion. In short, Python is a good language for Getting Things Done with a minimum of fuss. I use it frequently in my own work, and many readers will find it sufficient for much or all of their own programming.

Mathematically, I start at a rather elementary level: the book assumes no mathematical background beyond algebra and logarithms. In a few places I use examples from elementary calculus, but a reader who has not studied calculus can skip these examples. I don't assume a previous course in discrete mathematics; I introduce concepts from discrete mathematics as I go along. Some of these are simple but powerful concepts that (unfortunately) some

programmers never learn, and we'll see how to use them to create simple and elegant solutions to programming problems.

For example, one recurring theme in the book is the concept of a monoid. It turns out that monoids (more than, for example, groups and semigroups) are ubiquitous in the data types and data structures that programmers use most often. I emphasize the extent to which all monoids behave alike and how concepts and algorithms can be transferred from one to another.

I recommend this book for use in a first university-level course, or even an advanced high-school course, for mathematically-oriented students who have had some exposure to computers and programming. For students with no such exposure, the book could be supplemented by an introductory programming textbook, using either Python or another programming language, or by additional lecture material or tutorials presenting techniques of programming. Or the book could be used in a second course that is preceded by an introductory programming course of the usual kind.

Otherwise, the ideal reader is someone who has had at least some experience with programming, using either Python or another programming language. In fact, I hope that some of my readers will be quite experienced programmers who may never have been through a modern, mathematically-oriented program of study in computer science. If you are such a person, you'll see many ideas that will probably be new to you and that will probably improve your programming.

At the end of most chapters is a set of exercises. Instructors can use these exercises in laboratory sessions or as homework exercises, and some can be used as starting points for class discussions. Many instructors will want to supplement these exercises with their own extended programming assignments.

In a number of places I introduce a topic and then say something like "... details are beyond the scope of this book." The book could easily expand to encompass most of the computer science curriculum, and I had to draw the line somewhere. I hope that many readers, especially students, will pursue some of these topics further, perhaps with the aid of their instructors or in later programming and computer science classes. Some of the topics are

exception handling, parallel computing, distributed computing, various advanced data structures and algorithms, object-oriented programming, and state machines.

Similarly, I could have included many more topics in discrete mathematics than I did, but I had to draw the line somewhere. Some computer scientists and mathematicians may well disagree with my choices, but I have tried to include topics that have the most relevance to day-to-day programming. If you are a computer science student, you will probably go on to study discrete mathematics in more detail, and I hope that the material in this book will show you how the mathematics is relevant to your programming work and motivate you to take your discrete-mathematics classes more seriously.

This book is not designed to be a complete textbook or reference manual for the Python language. The book will introduce many Python constructs, and I'll describe them in enough detail that a reader unfamiliar with Python should be able to understand what's going on. However, I won't attempt to define these constructs in all their detail or to describe everything that a programmer can do with them. I'll omit some features of Python entirely: they are more advanced than we'll need or are otherwise outside the scope of this book. Here are a few of them:

- some types, such as `complex` and `byte`
- some operators and many of the built-in functions and methods
- string formatting and many details of input and output
- the extensive standard library and the many other libraries that are commonly available
- some statement types, including `break` and `continue`, and else-clauses in while-statements and for-statements
- many variations on function parameters and arguments, including default values and keyword parameters
- exceptions and exception handling

-
- almost all “special” attributes and methods (those whose names start and end with a double underbar) that expose internal details of objects
 - many variations on class definitions, including multiple inheritance and decorators

Any programmer who uses Python extensively should learn about all of these features of the language. I recommend that such a person peruse a comprehensive Python textbook or reference manual.²

In any case, there is more to Python than I present in this book. So whenever you think to yourself, “I see I can do x with Python — can I do y too?”, maybe you can. Again, you can find out in a Python textbook or reference manual.

This book will describe the most modern form of Python, called Python 3. It may be that the version of Python that you have on your computer is a version of Python 2, such as Python 2.3 or 2.7. There are only a few differences that you may see as you use the features of Python mentioned in this book. Here are the most important differences (for our purposes) between Python 3 and Python 2.7, the final and most mature version of Python 2:

- In Python 2, `print` is a statement type and not a function. A `print` statement can contain syntax not shown in the examples in this book; however, the syntax used in the examples — `print(e)` where e is a single expression — works in both Python 2 and Python 3.
- Python 2 has a separate “long integer” type that is unbounded in size. Conversion between plain integers and long integers (when necessary) is largely invisible to the programmer, but long integers are (by default) displayed with an “L” at the end. In Python 3, all integers are of the same type, unbounded in size.
- Integer division produces an integer result in Python 2, not a floating-point result as in Python 3.

² As of the time of writing, comprehensive Python documentation, including the official reference manual, can be found at <http://docs.python.org>.

-
- In Python 2, characters in a string are ASCII and not Unicode by default; there is a separate Unicode type.

Versions of Python earlier than 2.7 have more incompatibilities than these: check the documentation for the version you use.

In the chapters that follow I usually use the author's “we” for a first-person pronoun, but I say “I” when I am expressing my personal opinion, speaking of my own experiences, and so on. And I follow this British punctuation convention: punctuation is placed inside quotation marks only if it is part of what is being quoted. Besides being more logical (in my opinion), this treatment avoids ambiguity. For example, here's how many American style guides tell you to punctuate:

To add one to `i`, you would write “`i = i + 1.`”

Is the “.” part of what you would write, or not? It can make a big difference, as any programmer knows. There is no ambiguity this way:

To add one to `i`, you would write “`i = i + 1`”.

I am grateful to all the friends and colleagues who have given me help, suggestions, and support in this writing project, most prominently Lisa Beinhoff, Horst Clausen, Jeff Havlena, Peter Henderson, Daryl Lee, Subhashish Mazumdar, Angelica Perry, Steve Schaffer, John Shipman, and Steve Simpson.

Finally, I am pleased to acknowledge my debt to a classic textbook: *Structure and Interpretation of Computer Programs* (SICP) by Abelson and Sussman.³ I have borrowed a few ideas from it: in particular, for my treatments of higher-order functions and streams. And I have tried to make my book a showcase for Python much as SICP was a showcase for the Scheme language. Most important, I have used SICP as an inspiration, a splendid example of how programming can be taught when educators take programming seriously.

³Harold Abelson and Gerald Jay Sussman with Julie Sussman, *Structure and Interpretation of Computer Programs* (Cambridge, Mass.: The MIT Press, 1985).

Chapter 1

Introduction

1.1. Programs, data, and mathematical objects

A master programmer learns to think of programs and data at many levels of detail at different times. Sometimes the appropriate level is bits and bytes and machine words and machine instructions. Often, though, it is far more productive to think and work with higher-level data objects and higher-level program constructs.

Ultimately, at the lowest level, the program code that runs on our computers is patterns of bits in machine words. In the early days of computing, all programmers had to work with these machine-level instructions all the time. Now almost all programmers, almost all the time, use higher-level programming languages and are far more productive as a result.

Similarly, at the lowest level all data in computers is represented by bits packaged into bytes and words. Most beginning programmers learn about these representations, and they should. But they should also learn when to rise above machine-level representations and think in terms of higher-level data objects.

The thesis of this book is that, very often, mathematical objects are exactly the higher-level data objects we want. Some of these mathematical objects are numbers, but many — the objects of discrete mathematics — are quite different, as we will see.

So this book will present programming as done at a high level and with a mathematical slant. Here's how we will view programs and data:

Programs will be text, in a form (which we call “syntax”) that does not look much like sequences of machine instructions. On the contrary, our programs will be in a higher-level programming language, whose syntax is designed for

writing, reading, and understanding by humans. We will not be much concerned with the correspondence between programming-language constructs and machine instructions.

The data in our programs will reside in a computer's main storage (which we often metaphorically call “memory”) that may look like a long sequence of machine words, but most of the time we will not be concerned with exactly how our data objects are represented there; the data objects will look like mathematical objects from our point of view. We assume that the main storage is quite large, usually large enough for all the data we might want to put into it, although not infinite in size.

We will assume that what looks simple in a program is also reasonably simple at the level of bits and bytes and machine instructions. There will be a straightforward correspondence between the two levels; a computer science student or master programmer will learn how to construct implementations of the higher-level constructs from low-level components, but from other books than this one. We will play fair; we will not present any program construct that hides lengthy computations or a mass of complexity in its low-level implementation. Thus we will be able to make occasional statements about program efficiency that may not be very specific, but that will at least be meaningful. And you can be assured that the programming techniques that we present will be reasonable to use in practical programs.

The language of science is mathematics; many scientists, going back to Galileo, have said so. Equally, the language of computing is mathematics. Computer science education teaches why and how this is so, and helps students gain some fluency in the aspects of this language that are most relevant to them. The current book takes a few steps in these directions by introducing some of the concepts of discrete mathematics, by showing how useful they can be in programming, and by encouraging programmers to think mathematically as they do their work.

1.2. A first look at Python

For the examples in this book, we'll use a particular programming language, called Python. I chose Python for several reasons. It's a language that's in common use today for producing many different kinds of software. It's available for most computers that you're likely to use. And it's a clean and well-designed language: for the most part, the way you express things in Python is straightforward, with a minimum of extraneous words and punctuation. I think you're going to enjoy using Python.

Python falls into several categories of programming language that you might hear programmers talk about:

- It's a scripting language. This term doesn't have a precise definition, but generally it means a language that lends itself to writing little programs called *scripts*, perhaps using the kinds of commands that you might type into a command-line window on a typical computer system. For example, some scripts are programs that someone writes on the spur of the moment to do simple manipulations on files or to extract data from them. Some scripts control other programs, and system administrators often use scripting languages to combine different functions of a computer's operating system to perform a task. We'll see examples of Python scripts shortly. (Other scripting languages that you might encounter are Perl and Ruby.)
- It's an object-oriented language. Object orientation is a very important concept in programming languages. It's a rather complex concept, though, so we'll wait to discuss it until Chapter 11. For now, let's just say that if you're going to be working on a programming project of any size, you'll need to know about object-oriented programming and you'll probably be using it. (Other object-oriented languages are Java and C++.)
- It's a very high-level language, or at least it has been called that. This is another concept that doesn't have a precise definition, but in the case of Python it means that mathematical objects are built into the core of the language, more so than in most other programming languages. Furthermore, in many cases we'll be able to work with these objects in notation that

resembles mathematical notation. We'll be exploiting these aspects of Python throughout the book.

Depending on how you use it, Python can be a language of any of these kinds or all of them at once.

Let's look at a few simple Python programs, to give you some idea of what Python looks like.

The first program is the kind of very short script that a Python programmer might write to use just once and then discard. Let's say that you have just attended a lecture, and you met someone named John, but you can't remember his last name. Fortunately, the lecturer has a file of the names of all the attendees and has made that file available to you. Let's say that you have put that file on your computer and called it “names”. There are several hundred names in the file, so you'd like to have the computer do the searching for you. Example 1.1 shows a Python script that will display all the lines of the file that start with the letters “John”.

Example 1.1. Finding a name

```
file = open("names")
for line in file:
    if line.startswith("John"):
        print(line)
```

You may be able to guess (and guess correctly) what most of the parts of this script do, especially if you have done any programming in another programming language, but I'll explain the script a line at a time. Let's not bother with the fine points, such as what the different punctuation marks mean in Python; you'll learn all that later. For now, I'll just explain each line in very general terms.

```
file = open("names")
```

This line performs an operation called “opening” a file on our computer. It's a rather complicated sequence of operations, but the general idea is this: get a file named “names” from our computer's file system and make it available for our program to read from. We give the name `file` to the result.

Here and in the other examples in this book, we won't worry about what might happen if the `open` operation fails: for example, if there is no file with the given name, or if the file can't be read for some reason. Serious Python programmers need to learn about the features of Python that are used for handling situations like these, and need to include code for handling exceptional situations in most programs that do serious work.¹ In a simple one-time script like this one, though, a Python programmer probably wouldn't bother. In any case, we'll omit all such code in our examples, simply because that code would only distract from the points that we are trying to make.

```
for line in file:
```

This means, “For each line in `file`, do what comes next.” More precisely, it means this: take each line of `file`, one at a time. Each time, give that line the name `line`, and then do the lines of the program that come next, the lines that are indented.

```
    if line.startswith("John"):
```

This means what it appears to mean: if `line` starts with the letters “John”, do what comes next.

```
        print(line)
```

Since the line of the file starts with “John”, it's one that we want to see, and this is what displays the line. On most computers, we can run the program in a window on our computer's screen, and `print` will display its results in that window.

¹The term for such code is “exception handling”, in case you want to look up the topic in Python documentation. Handling exceptions properly can be complicated, sometimes involving difficult design decisions, which is why we choose to treat the topic as beyond the scope of the current book.

If you run the program, here's what you might see (depending, of course, on what is in the file names).

```
John Atencio
John Atkins
Johnson Cummings
John Davis
John Hammerstein
```

And so on. This is pretty much as you might expect, although there may be a couple of surprises here. Why is this output double-spaced? Well, it turns out that each line of the file ends with a “new line” character, and the `print` operation adds another. (As you learn more details of Python, you'll probably learn how to make output like this come out single-spaced if that's what you'd prefer.) And why is there one person here with the first name “Johnson” instead of “John”? That shouldn't be a surprise, since our simple little program doesn't really find first names in a line: it just looks for lines in which the first four letters are “John”. Anyway, this output is probably good enough for a script that you're only going to use once, especially if it reminds you that the person you were thinking of is John Davis.

Now let's say that you'd like to get in touch with John Davis. Your luck continues: the lecturer has provided another file containing the names and email addresses of all the attendees. Each line of the file contains a person's name and that person's email address, separated by a comma.

Suppose you transfer that file to your computer and give it the name “`emails`”. Then Example 1.2 shows a Python script that will find and display John Davis's email address if it's in the file.

Example 1.2. Finding an email address

```
file = open("emails")
for line in file:
    name, email = line.split(",")
    if name == "John Davis":
        print(email)
```

Let's look at this program a line or two at a time.

```
file = open("emails")
for line in file:
```

These lines are very much like the first two lines of the previous program; the only difference is the name of the file in the first line. In fact, this pattern of code is common in programs that read a file and do something with every line of it.

```
    name, email = line.split(",")
```

The part `line.split(",")` splits `line` into two pieces at the comma. The result is two things: the piece before the comma and the piece after the comma. We give the names “`name`” and “`email`” to those two things.

```
    if name == "John Davis":
```

This says: if `name` equals (in other words, is the same as) “`John Davis`”, do what comes next. Python uses “`==`”, two adjacent equals-signs, for this kind of comparison. You might think that just a single equals-sign would mean “equals”, but Python uses “`=`” to associate a name with a thing, as we have seen. So, to avoid any possible ambiguity, Python uses a different symbol for comparing two things for equality.

```
        print(email)
```

This displays the result that we want.

As our final example, let's take a very simple computational task: finding the average of a collection of numbers. They might be a scientist's measurements

of flows in a stream, or the balances in a person's checking account on different days, or the weights of different boxes of corn flakes. It doesn't matter what they mean: for purposes of our computation, they are just numbers.

Let's say, for the sake of the example, that they are temperatures. You have a thermometer outside your window, and you read it at the same time each day for a month. You record each temperature to the nearest degree, so all your observations are whole numbers (the mathematical term for these is “integers”). You put the numbers into a file on your computer, perhaps using a text-editing or word-processing program; let's say that the name of the file is “observations”. At the end of the month, you want to calculate the average temperature for the month.

Example 1.3 shows a Python program that will do that computation. It's a little longer than the previous two programs, but it's still short and simple enough that we might call it a “script”.

Example 1.3. Average of a collection of observations

```
sum = 0
count = 0

file = open("observations")
for line in file:
    n = int(line)
    sum += n
    count += 1

print(sum/count)
```

Let's look at this program a line or two at a time.

```
sum = 0
count = 0
```

To compute the average of the numbers in the file, we need to find the sum of all the numbers and also count how many there are. Here we give the names

`sum` and `count` to those two values. We start both the sum and the count at zero.

```
file = open("observations")
for line in file:
```

As in the previous two programs, these lines say: open the file that we want and then, for each line of the file, do what comes next. Specifically, do the lines that are indented, the next three lines.

```
    n = int(line)
```

A line of a file is simply a sequence of characters. In this case, it will be a sequence of digits. The program needs to convert that into a single thing, a number. That's what `int` does: it converts the sequence of digits into an integer. We give the name `n` to the result.

```
    sum += n
    count += 1
```

In Python, “+=” means “add the thing on the right to the thing on the left”. So, “`sum += n`” means “add `n` to `sum`” and “`count += 1`” means “add 1 to `count`”. This is the obvious way to accumulate the running sum and the running count of the numbers that the program has seen so far.

```
print(sum/count)
```

This step is done after all the numbers in the file have been summed and counted. It displays the result of the computation: the average of the numbers, which is `sum` divided by `count`.

Notice, by the way, that we've used blank lines to divide the lines of the program into logical groups. You can do this in Python, and programmers often do. This doesn't affect what the program does, but it might make the program a little easier for a person to read and understand.

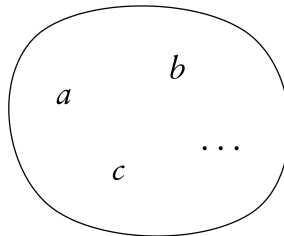
So now you've seen three very short and simple Python programs. They aren't entirely typical of Python programs, though, because they illustrate only a few of the most basic parts of the Python language. Python has many more

features, and you'll learn about many of them in the remaining chapters of this book. But these programs are enough examples of Python for now.

1.3. A little mathematical terminology

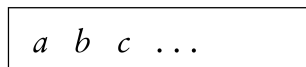
Now we'll introduce a few mathematical terms that we'll use throughout the book. We won't actually be *doing* much mathematics, in the sense of deriving formulas or proving theorems; but, in keeping with the theme of the book, we'll constantly use mathematical terminology as a vocabulary for talking about programs and the computational problems that we solve with them.

The first term is *set*. A set is just an unordered collection of different things.



For example, we can speak of the set of all the people in a room, or the set of all the books that you have read this year, or the set of different items that are for sale in a particular shop.

The next term is *sequence*. A sequence is simply an ordered collection of things.



For example, we can speak of the sequence of digits in your telephone number or the sequence of letters in your surname. Unlike a set, a sequence doesn't have the property that all the things in it are necessarily different. For example, many telephone numbers contain some digit more than once.

You may have heard the word “set” used in a mathematical context, or you may know the word just from its ordinary English usage. It may seem strange to call the word “sequence” a mathematical term, but it turns out that

sequences have some mathematical properties that we'll want to be aware of. For now, just notice the differences between the concepts “set” and “sequence”.

Let's try applying these mathematical concepts to the sample Python programs that we've just seen. In each of them, what kind of mathematical object is the data that the program operates on?

First, notice that each program operates on a file. A file, at least as a Python program sees it, is a sequence of lines. Code like this is very common in Python programs that read files a line at a time:

```
file = open("observations")
for line in file:
```

In general, the Python construct

for element in sequence :

is the Python way to do something with every element of a sequence, one element at a time.

Furthermore, each line of a file is a sequence of characters, as we have already observed. So we can describe a file as a sequence of sequences.

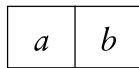
But let's look deeper.

Let's take the file of names in our first example (Example 1.1). In terms of the information that we want to get from it, the file is a collection of names. What kind of collection? We don't care about the order of names in it; we just want to see all the names that start with “John”. So, assuming that our lecturer hasn't included any name twice by mistake, the collection is a set as far as we're concerned.

In fact, both the input and the output of this program are sets. The input is the set of names of people who attended the lecture. The output is the set of members of that input set that start with the letters “John”. In mathematical terminology, the output set is a subset of the input set.

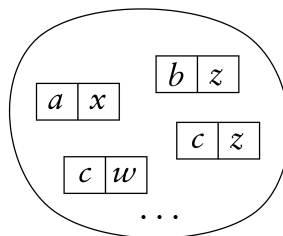
Notice that the names in the input may actually have some ordering; the point is that we don't care what the ordering is. For example, the names may be in alphabetical order by surname; we might guess that from the sample of the output that we have seen. And of course the names have an ordering imposed on them simply from being stored as a sequence of lines in a file. The point is that any such ordering is irrelevant to the problem at hand and what we intend to do with the collection of names.

What about the file of names and email addresses in the second example (Example 1.2)? First, let's consider the lines of that file individually. Each line contains a name and an email address. In mathematical terminology, that data is an *ordered pair*.

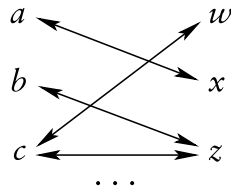


The two elements are “ordered” in the sense that it makes a difference which is first and which is second. In this case, it makes a difference because the two elements mean two different things. An ordered pair is not the same as a set of two things.

Now what about the file as a whole? Like the input file for the first program, it is a set. We don't care whether the file is ordered by name or by email address, or not ordered at all; we just want to find an ordered pair in which the first element is “John Davis” and display the corresponding second element. So (again assuming that the file doesn't contain any duplicate lines) we can view the input data as a set of ordered pairs.

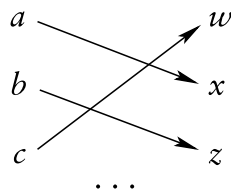


There's another mathematical name for a set of ordered pairs: a *relation*. We can view a set of ordered pairs as a mathematical structure that relates pairs of things with each other.



In the current program, our input data relates names with corresponding email addresses.

One particular kind of relation will be very important to us: the *mapping*. This is a set of ordered pairs in which no two first elements are the same. We can think of a mapping as a structure that's like a mechanism for looking things up: give it a value (such as a name) and, if that value occurs as a first element in the mapping, you get back a second element (such as an email address).



In mathematics, another word for “mapping” is “function”; you probably know about mathematical functions like the square-root and trigonometric functions. The word “function” has other connotations in programming, though, so we'll usually use the word “mapping” for the mathematical concept.

We don't know whether the input data for this program is a mapping. Some attendees may have given the lecturer more than one email address, and the lecturer may have included them all in the file. In that case, the data may contain more than one ordered pair for some names, and so the data is a relation but not a mapping. But if the lecturer took only one email address per person, the data is not only a relation but a mapping. We may not care

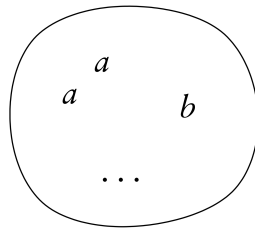
about the distinction in this case: if our program displays more than one email address for John Davis, that's probably OK.

Now what about the input data in our third example, the program that computes an average (Example 1.3)? When you add up a group of numbers to average them, the order of the numbers and the order of the additions don't matter. This fact follows from two fundamental properties of integer addition:

- The *associative property*: $(a + b) + c = a + (b + c)$ for any integers a , b , and c
- The *commutative property*: $a + b = b + a$ for any integers a and b

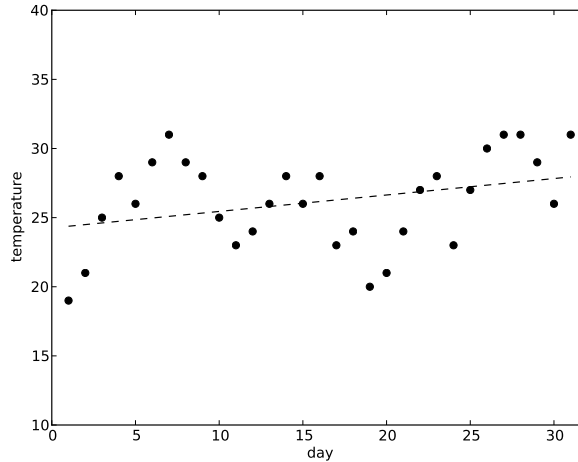
So is this data a set, like the input data for our first two programs? Not quite. The difference is that the same number may appear in the input data more than once, because the temperature outside your window may be the same (to the nearest degree) on two different days.

This data is a *multiset*. This is a mathematical term for a collection that is like a set, except that its elements are not necessarily all different: any element may occur in the multiset more than once. But a multiset, like a set, is an unordered collection.



So, if our task is to compute the average of our collection of observations, the term “multiset” is an accurate description of that collection. But suppose we wanted to plot our observations to show the trend of the temperatures over the month, as in Figure 1.1?

Figure 1.1. Temperatures over a month, with trend line



Then the order of the observations would be important, so we would view the data as a sequence. The point here is that what kind of mathematical object a collection of data is, from a programmer's point of view, depends not only on properties of the data but also on what the programmer wants to do with the data.

Let's summarize. Here are the kinds of mathematical objects that we've mentioned so far:

- set
- multiset
- sequence
- ordered pair
- relation
- mapping

And here are the instances of these mathematical objects that we've observed in our examples:

- A generic file: a *sequence of sequences*.
- The data for our first script (names, Example 1.1): a *set*.
- The data for our second script (emails, Example 1.2): a *set of ordered pairs*, forming a *relation*, possibly a *mapping*.
- The data for our third script (average of observations, Example 1.3): a *multiset*. But if we want to use the same data to plot a trend, a *sequence*.

In later chapters we'll explore properties of these and other mathematical objects and their connections with the data and computations of programs. Meanwhile, let's conclude the chapter with two simple observations:

- A collection of data, as a programmer views it, is likely to be a set, or a sequence, or a mapping, or some other well-known mathematical object.
- You can often view a collection of data as more than one kind of mathematical object, depending on what you want to do with the data.

Terms introduced in this chapter

script

set

sequence

ordered pair

relation

mapping

associative property

commutative property

multiset

Chapter 2

An overview of Python

2.1. Introduction

In this chapter and the two chapters that follow we'll give an overview of the Python language, enough (we hope) for you to understand the Python examples in the rest of the book. Along the way, we'll introduce terms for many Python concepts. Most of these terms are not specific to Python, but are part of the common language for speaking about programming languages in the computer science community. We'll use these terms in many places later in the book; so, even if you already know Python, you might want to skim through the chapters just to make sure that you are familiar with all the terms.

You have seen examples of Python programs that take their input data from files. In fact, Python programs themselves are made of text in files. If you had the scripts of the previous chapter on your computer, each would be in a file of its own.

On most computer systems, you can also use the Python interpreter interactively. You can type small bits of Python to the interpreter, and it will display the results. For example, you can use the interpreter as a calculator. You can type

```
3 + 2
```

and the interpreter will display

```
5
```

If you have your computer handy as you are reading this book, you might like to experiment with the Python constructs that you are reading about. Go ahead and type pieces of Python to the interpreter and see what happens. Feel free to do this at any time as you read.

2.2. Values, types, and names

We'll start by explaining some very basic concepts of Python; you might think of these as defining how to think about computation in the Python world.

Computation in Python is done with *values*. Computations operate on values and produce other values as results. When you type “3 + 2” to the Python interpreter, Python takes the values 3 and 2, performs the operation of addition on them, and produces the value 5 as a result.

We have seen a few kinds of values already. The numbers 3 and 2 are values. So are sequences of characters like “John”. So are more complicated things like the opened files that the scripts of Chapter 1 used. As we'll see in later chapters, Python values also include other kinds of sequences, as well as sets and mappings and many other kinds of things.

Every value has a *type*. The type of whole numbers like 3 and 2 is “integer”. The type of a sequence of characters is “string”. There are other types in Python, and we'll see many of them later in the book.

A type is more than just a Python technical term for a kind of value. A type has two important properties, besides its name: a set of possible values, and the set of operations that you can do with those values. In Python, one of the operations on integers is addition, and one of the operations on strings is splitting at a given character.

Finally, *names* are central to Python programming, and they are used in many ways. One important example is this: we can *bind* a name to a value. That's what happened when we did this in our script that computed an average:

```
count = 0
```

This binds the name `count` to the value 0. Another name for the operation in this example is *assignment*: we say that we assign 0 to `count` and if a line like this occurs in a Python program, the line is called an *assignment statement*. There are several ways to bind a name to a value in Python, but the assignment statement is the most basic way.

A name that is bound in an assignment statement like this is called a *variable*. That's because the binding is not necessarily permanent: the name can be bound to a different value later. For example, our script for computing an average has this line farther along:

```
count += 1
```

This binds `count` to a new value, which is one greater than the value that it was bound to before. A program can do such rebinding many times, and in fact our averaging script does: once for each line of the file of observations.

In Python you could even do this in the same program, although you probably wouldn't want to:

```
count = "Hello, John!"
```

So even the type of a variable isn't permanent. In Python, a type is a property of a value. We can speak of the type of a variable, but that's just the type of the value that happens to be bound to that variable.

However, as Python is actually used in practice, most variables have the same type throughout a program — that just turns out to be the most natural way for programmers to use variables. Furthermore, good programmers choose names that have some logical relationship with the way that the names are used. Giving a name like `count` to a string like `"Hello, John!"` is just silly, even though the Python interpreter would let you do it.

Saying that an assignment binds a name to a value is actually a slight oversimplification in Python. What actually happens is that a name is bound to something called an “object”, and it's the object that has a value. Later in the book we'll see how the distinction can make a difference. For now, this fine point isn't important, and we can just think of the name as being bound directly to the value.

2.3. Integers

Python integers are like mathematical integers: that is, they are the positive and negative whole numbers and zero.

Some of the operations that you can do with integers in Python, as you might expect, are addition, subtraction, multiplication, and division. Each of these operations is denoted by a symbol that comes between the quantities operated on, like this:

```
3 + 2
3 - 2
3 * 2
3 / 2
```

Each of those symbols is called an *operator*, and the quantities that they operate on are called *operands*. Many operators, like these, operate on two operands; such operators are called *binary* operators. (The term “binary”, when used in this way, doesn't have anything to do with the binary number system.)

In a Python program, a sequence of one or more digits, like 123 or 2 or 0, represents a non-negative integer value. The sequence of digits is an example of a *constant*: a symbol that can't be bound to anything other than the value that it represents.

To get a negative integer, we precede a sequence of digits with a minus sign in the obvious way, like this:

```
-123
```

Here, the minus sign is a *unary* operator, meaning that it takes only one operand. Notice that `-` can be either a binary operator or a unary operator, depending on context.

The combination of an operator and its operands is called an *expression*. As in most programming languages, larger expressions can be built up by combining smaller expressions with operators, and parentheses can be used for grouping. We speak of *evaluating* an expression: this means finding the values of all its parts and performing all the operations to obtain a value.

If the operands of the `+`, `-`, and `*` operators are integers, the result is also an integer. The operator `/` is different: for example, the result of evaluating the expression `3 / 2` is 1.5 just as in mathematics. But that 1.5 is not an integer,

of course, because it has a fractional part. It's an example of a *floating-point* number; we'll explain numbers of that type in the next section.

In Python, the result of dividing two integers with the `/` operator is always a floating-point number, whether the division comes out even or not. There's another division operator that always gives an integer result: it is `//`. So, for example, the value of `4 / 2` is the floating-point number `2.0`, but the value of `4 // 2` is `2`, an integer. If the result of `//` does not come out even, it is rounded down, so that the value of `14 // 3` is the integer `4`, with no fractional part; the remainder, `2`, is dropped.

The operator `%` gives the remainder that would be left after a division. The value of `14 % 3` is `2`, for example.

Python has other integer operators. Here's one more: `**` is the exponentiation operator. For example, `2 ** 8` means 2^8 .

Python also has operators that combine arithmetic and assignment: these are called *augmented assignment* operators. We have already seen one of these, the `+=` operator, in this augmented-assignment statement:

```
count += 1
```

In evaluating this statement, the Python interpreter does the `+` operation in the usual way and then binds the left operand to the result. Python also has `-=` and `*=` and so on, but the `+=` operator is probably the one that programmers use most often.

In Python, there is no intrinsic limit on the size of an integer. The result of evaluating the expression `2**2**8` is a rather large number, but it's a perfectly good integer in Python, and Python integers can be much larger than that one.

However, there are practical limits to the size of a Python integer. You can't evaluate the expression `2**2**100` using the Python interpreter. That value would contain far more digits than your computer can store, and even if the Python interpreter could find somewhere to put all the digits, evaluating the expression would take an enormously long time.

In mathematics, $2^{2^{100}}$ is a perfectly good number. It's a finite number, and it isn't hard to denote finite numbers that are much larger than that one: think of raising 2 to *that* power, for example. In mathematics there are also ways of making sense of infinite numbers, and mathematics draws a distinction between all finite numbers and the infinite numbers.¹

Unlike many programming languages, Python has a way of representing “infinity”, as we'll see in the next section. There is only one infinite number (and its negative) in Python, and the Python “infinity” has rather limited usefulness in Python programs, but it does exist.

In Python programming, and in all programming for that matter, it's important to recognize a third category of numbers, besides finite and infinite: numbers that are finite but that are far too large to compute with in practice, such as $2^{2^{100}}$.

As we'll see in later chapters, sets can be represented as Python values, and so can sequences, mappings, and other mathematical structures. As with integers, the Python values are similar to the mathematical objects; Python was designed so that they would be. That's good, because we can use mathematical thinking to describe and understand those values and the operations on them. But, as with integers, Python sets and sequences and the rest have practical limitations. For example, a Python programmer must avoid computations that would try to construct sets that are far too large to store or operate on.

So, whereas in mathematics there's a distinction between finite and infinite, in programming there's also a distinction between finite and finite-but-far-too-large. The distinction applies to computations too: there are many computational problems that can't be solved in practice because the computations would take far too long. Drawing a line between practical and impractical, and categorizing values and computations according to which side of the line they are on, are central issues in the field of computer science, as you will see later in your studies if you are a computer science student.

¹Yes, in mathematics there are different infinite numbers. For example, the number of points on a line is greater than the number of integers: infinitely greater, in fact.

2.4. Floating-point numbers

In Python, the floating-point numbers are another type, whose name is "float". Python uses floating-point numbers to represent numbers with a fractional part, and produces them in computations that may not come out even, like division of two integers using the `/` operator. As Python integers represent mathematical integers, Python floating-point numbers represent the real numbers of mathematics. Floating-point numbers are only an approximate representation of the real numbers, though; they differ from mathematical real numbers in some important ways, as we will see.

Python has two kinds of floating-point constants. The first is simply a sequence of digits with a decimal point somewhere in it; `3.14` and `.001` are examples. The second is a version of “scientific notation”: the number 6.02×10^{23} can be written in Python as `6.02E23`. The number after the “E” (you can also use “e”) is the power (“exponent”) of ten used in the scientific notation; it can be negative, as in `1e-6`, which represents one millionth.

Floating-point arithmetic in Python is much like integer arithmetic, except that if the operands in an expression are floating-point, so is the result. Floating-points and integers can be mixed in an expression: if one operand of a binary operator is a floating-point and the other is an integer, the integer is converted to a floating-point value and then the operation is done, giving a floating-point value.

To convert explicitly from an integer to a floating-point number or vice versa, we can use a Python *function*. As we'll see in later chapters, Python functions can behave in several different ways, but the kind that we'll consider now is like a mathematical function. It's a mapping: it takes a value, called an *argument*, and produces another value as a function of the argument.

To use a function, we write a *function call*, which is another kind of expression besides those that we have seen. In the form that we'll consider now, a function call consists of the name of a function, followed by an expression in parentheses; that expression is the argument. When the Python interpreter evaluates a function call, we say that it *calls* the function, *passing* the argument

to the function. The function *returns* a value, which becomes the value of the function call expression.

To convert an integer value to floating-point, we can use the `float` function. For example, if the variable `n` has the value 3, then `float(n)` has 3.0 as a value. To convert from floating-point to integer, we can use the `int` function if we just want to truncate the fractional part, or the `round` function if we want to round to the nearest integer. For example, if the variable `x` has the value 2.7, the value of `int(x)` is 2 and the value of `round(x)` is 3.

We can also use `float` to create a Python representation of “infinity”. Python has no constant that represents “infinity”, but we can create the Python value that represents “infinity” by writing `float("inf")` or `float("Infinity")` (the string passed to `float` can have any combination of upper-case and lower-case letters). We can get the negative infinity by writing `float("-inf")` or `-float("Infinity")` or similar expressions.

Probably the most useful property of the Python “infinity” is that it is a floating-point number that is greater than any other floating-point number and greater than any integer. Similarly, the Python “negative infinity” is a floating-point number that is less than any other floating-point number and less than any integer. Except for showing a few applications of those properties, we won't say much more about Python's positive and negative infinity in this book.

Floating-point numbers (except for positive and negative infinity) are stored in the computer using a representation that is much like scientific notation: with a sign, an exponent, and some number of significant digits. On most computers, Python uses the “double precision” representation provided by the hardware.

For our purposes here, the exact representation isn't important, except for one point: both the exponent and the significant digits are represented using a fixed number of bits. An interesting consequence of this is that only finitely many floating-point numbers are representable on any given computer. In fact, in Python there are many more integers than floating-point numbers! This is the opposite of the situation in mathematics.

Notice that there must be a limit to the magnitude of a floating-point number, since there's an upper bound to the value of the exponent. This limitation usually isn't serious in practice, though, since on modern computers a floating-point number can easily be large enough for almost all common uses, such as representing measurements in the physical universe.

A more important limitation is that a floating-point number contains only a limited number of significant digits. This means that most of the mathematical real numbers can be represented only approximately with floating-point numbers. It also means that the result of any computation that produces a floating-point result, such as evaluating the expression $1/3$, will be truncated or rounded to a fixed number of significant digits, giving only an approximation to the true mathematical value in most cases.

Thus, we must be careful when we compute with floating-point numbers, keeping in mind that they are only approximations. For example, we can't assume that the value of $1/3 + 1/3 + 1/3$ is exactly equal to 1.0 ; we can only assume that the two values are approximately equal. The difference between a floating-point result and the true mathematical value is called roundoff error; in some situations, especially in long computations, roundoff error can build up and cause computations to be unacceptably inaccurate.

2.5. Strings

As we have already mentioned, sequences of characters are called *strings*. In Python, a string can contain not only the characters available on your keyboard, but all the characters of the character set called “Unicode”. Unicode contains characters from most of the world's written languages, including Chinese, Arabic, Hindi, and Cherokee, to name just a few. Unicode also contains mathematical symbols, technical symbols, unusual punctuation marks, and many more characters. For our purposes in this book, though, the characters on your keyboard will be enough.

A string constant is any sequence of characters enclosed in double-quote characters, as in

```
"Here's one"
```

or single-quote characters, as in

```
'His name is "John"'
```

Notice how a string delimited by double-quote characters can contain single-quote characters and vice versa.

The sequence of characters in a string may be empty, as in `""`; this string is called the *empty string* or the “null string”. Yes, the empty string does have uses, as we will see.

Python has no “character” type; it treats a single character as the same as a one-character string.

There is a function to convert from other types, such as integers and floating-point numbers, to strings: its name is `str`. For example, `str(3)` produces the string value `"3"`.

One important operation on strings is *concatenation*, meaning joining together end-to-end. Python uses the `+` operator for concatenation. For example, the value of the expression `"python" + "interpreter"` is

```
"pythoninterpreter"
```

So Python gives the `+` operator three different meanings that we've seen so far: integer addition, floating-point addition, and string concatenation. We say that `+` is *overloaded* with these three meanings.

Python also overloads `*` to give it a meaning when one operand is a string and the other is an integer: this means concatenating together a number of copies. For example, the value of `3 * "Hello"` is `"HelloHelloHello"`. The integer can be either the left or the right operand. But Python doesn't overload `+` or `*` for any imaginable combination of types. For example, Python doesn't allow `+` of a string and an integer. You might guess that the Python interpreter would convert the integer to a string and do concatenation, but it won't.

Terms introduced in this chapter

type	evaluating
name	floating-point
binding	augmented assignment
assignment	function
assignment statement	argument
variable	function call
integer	calling a function
operator	passing an argument
operand	return
binary operator	empty string
constant	concatenation
unary operator	overloading
expression	

Exercises

1. We said that 2^8 was a perfectly good Python value but that 2^{100} was far too large. For expressions of the form `2**2**n`, what values of n make the value of the expression too large to compute in practice? Experiment with your Python interpreter. Start with small values of n and then try larger values. What happens?
2. What happens if you actually try to evaluate `2**2**100` using your Python interpreter? Let it run for a long time, if necessary. Can you explain what you see?
3. Estimate how many decimal digits it would take to write out 2^{100} . Hint: logarithms. You can use your computer, if that will help.
4. The value of a comparison like `1.0 == 1` is either `True` or `False`. Experiment with your Python interpreter: you will probably get `True` for the value of `1.0 == 1`, for example. Try `1/3 + 1/3 + 1/3 == 1.0`; you may get `True` or `False`, depending on how the rounding is done on your computer. Try to

find some comparisons that should give `True` according to mathematical real-number arithmetic, but that give you `False` in Python on your computer.

5. Does concatenation of strings have the associative property, as addition of integers does? Does concatenation of strings have the commutative property?
6. Make an improvement to the script that reads a file and finds lines that begin with “John”. Change it so that it actually compares the first name on each line with the name “John”, so that (for example) it doesn't display lines starting with “Johnson”. You have already seen enough of Python that you should be able to guess how to do this. Assume that each line of the file contains just a first name, a single blank space, and a last name. Test your solution using the Python interpreter.

Chapter 3

Python programs

3.1. Statements

Now we'll take the concepts and Python constructs from the last chapter and see how they can be used in larger Python constructs, up to and including whole programs.

A Python program is a sequence of *statements*. To illustrate some of the kinds of Python statements, let's look again at one of the sample programs from Section 1.2.

```
sum = 0
count = 0

file = open("observations")
for line in file:
    n = int(line)
    sum += n
    count += 1

print(sum/count)
```

This program contains several statements. Some of them are assignment and augmented-assignment statements; each is on a line of its own.

If a statement is too long to fit on one line, it can be broken across lines by using the backslash character `\` followed by a line break, like this:

```
z = x**4 + 4 * x**3 * y + 6 * x**2 * y**2 \
    + 4 * x * y**3 + y**4
```

However, if the line break is inside bracketing characters such as parentheses, the backslash is not needed:

```
z = round(x**4 + 4 * x**3 * y + 6 * x**2 * y**2
          + 4 * x * y**3 + y**4)
```

The last line of the sample program is also a statement. As it happens, `print` is a function in Python. (Here, its argument is a floating-point number, but in the examples of Section 1.2 we also saw `print` being used to display strings. In fact, `print` is overloaded to work with these and many other Python types.)

An expression in Python, by itself, can be a statement, and the last line of the program is an example. For the expressions that we have considered until now, using one as a statement would make no sense; the Python interpreter would just evaluate it and do nothing with the result. But some expressions have *side effects*: evaluating them has some effect besides producing a value.

Some Python functions are designed to be used as statements. They aren't mappings at all, because they don't produce values; they only cause side effects. The `print` function is one of these. Its “side effect”, which is really its only effect, is to display a value.

Assignment statements, augmented assignment statements, and expression statements are called *simple statements*. The line starting with “`for`” and the three lines that follow it are an example of a *compound statement*: a statement with other statements inside it.

```
for line in file:
    n = int(line)
    sum += n
    count += 1
```

The first line of a compound statement is called its *header*. A header starts with a *keyword* that indicates what kind of compound statement it is, and ends with a colon. Python has a number of keywords that are used for special purposes like this. As it happens, both `for` and `in` are keywords; they are structural parts of the header. Keywords can't be used as names; you can't have a variable named “`for`” or “`in`”.

The remaining lines of a compound statement, called the *body* of the statement, are indented with respect to the header.

As we have already seen, programs can contain blank lines, which are not statements and have no effect on what the program does, but are strictly for

the benefit of human readers. Similarly, all serious programming languages provide for *comments*, so that a programmer can insert commentary and explanations into a program. In Python, a comment starts with the character “#” and continues to the end of the line. Here's the average-of-numbers script again with some comments added:

```
# A program to display the average of
# the numbers in the file "observations".
# We assume that the file contains
# whole numbers, one per line.

sum = 0
count = 0

file = open("observations")
for line in file:
    n = int(line)    # convert digits to an integer
    sum += n
    count += 1

print(sum/count)
```

3.2. Conditionals

In a sequence of statements, the Python interpreter normally executes the statements one after another in the order they appear. We say that the *flow of control* passes from one statement to the next in order.

Sometimes the flow of control needs to be different. For example, we often want certain statements to be executed only under certain conditions. A construct that causes this to happen is called a *conditional*.

The basic conditional construct in Python is a compound statement that starts with the keyword `if`. We call such a statement an *if-statement* for short. The most basic kind of if-statement has this form:

```
if condition :
    statements
```

Here's an example that we saw in Section 1.2:

```
if name == "John Davis":  
    print(email)
```

Most commonly, the condition in an if-statement header is a comparison, as in the example above. The result of a comparison is a value of the *Boolean* type. This type has only two values, `True` and `False`, and that's the way constants of the Boolean type are written. These values behave like the integers 1 and 0 in most contexts — in fact, Python considers them numeric values and you can do arithmetic with them — but the main use of Boolean values in Python is to control execution in if-statements (and in the while-statements that we'll see in the next section).

The comparison operator that means “does not equal”, like “≠” in mathematics, is written `!=` in Python. The operators `<` and `>` mean “is less than” and “is greater than”, as you might expect. For “is less than or equal to”, like “≤” in mathematics, Python uses `<=`, and similarly `>=` means “≥”.

One more comparison operator is `in`, which (for example) can be used to test whether a character is in a string. Here is an example, where `c` is a variable containing a character:

```
if c in "aeiou":  
    print("c is a vowel")
```

More generally, if the operands of `in` are strings, the operator tests whether the left operand is a *substring* of the right operand; in other words, whether the sequence of characters of the left operand occurs as a sequence of characters in the right operand. We'll see many more uses for the `in` operator in later chapters.

Python has operators whose operands are Booleans, too: `and`, `or`, and `not`. The `not` operator is unary, and produces `False` if the value of its operand is `True` and `True` if the value of its operand is `False`, as you might expect.

The `or` operator also means what it appears to mean, but Python evaluates it in a particular way. In evaluating an expression of the form `A or B`, the Python interpreter first evaluates `A`. If its value is `True`, the value of the whole expression is `True`; otherwise, the interpreter evaluates `B` and uses its value as

the value of the whole expression. The `and` operator is evaluated similarly: the interpreter evaluates the second operand only if the value of the first is `False`.

So, for the `or` and `and` operators, the interpreter doesn't just evaluate both operands and then do the operation, as it does with most operators. This can make a difference. Look at this example:

```
if y != 0 and x/y > 1:
```

If `y` is zero, the value of the whole Boolean expression is `False`, and the Python interpreter doesn't even evaluate `x/y > 1`. That's fortunate in this case, because dividing by zero is undefined in Python just as it is in ordinary arithmetic.

Python `if`-statements can have other forms, to create other kinds of control flow that occur commonly in programs. For example, often a program needs to do one thing if a condition is true and something else if the condition is false. The Python statement that does so has this form:

```
if condition :  
    statements  
else:  
    statements
```

This is a compound statement with more than one “clause”, each with its own header and body; the headers are aligned with each other. Here's a simple example:

```
if a > b:  
    print("I'll take a")  
else:  
    print("I'll take b")
```

To distinguish more than two cases and handle them differently, you can combine `if`-statements, as in this example:

```
if testScore > medianScore:
    print("Above average.")
else:
    if testScore == medianScore:
        print("Average.")
    else:
        print("Below average.")
```

Here we have one if-statement that contains another if-statement. By definition, a compound statement contains other statements, and these can be compound statements themselves.

When constructs contain other constructs of the same kind, we say that they are *nested*. Expressions are another example since, in an expression that contains an operator, the operands can be expressions themselves. Nested constructs appear in many places in programming languages.

The pattern of control flow in the above example is so common that Python provides a shorter way to do it. An if-statement can contain a clause beginning with `elif`, meaning “else if”, between the if-clause and the else-clause:

```
if condition :
    statements
elif condition :
    statements
else:
    statements
```

So the example above could be written more concisely like this:

```
if testScore > medianScore:
    print("above average.")
elif testScore == medianScore:
    print("average.")
else:
    print("below average.")
```

There can be more than one elif-clause, for computations in which there are more than three cases. The else-clause can be omitted whether or not there are elif-clauses. Thus, the general form of the if-statement in Python is an if-

clause, followed optionally by one or more `elif`-clauses, followed optionally by an `else`-clause.

3.3. Iterations

Another pattern of control flow is extremely common in programs: executing the same section of code many times. We call this *iteration*.

Some iterations execute a section of code over and over forever, or at least until the program is interrupted or the computer is shut down. But we'll concentrate on the kind of iteration that executes a section of code as many times as necessary, according to the circumstances, and then stops.

Here's an example that we've already seen (Example 1.1):

```
file = open("names")
for line in file:
    if line.startswith("John"):
        print(line)
```

The compound statement starting with the keyword `for` is a Python iteration construct, the *for-statement*. As we have seen, this `for`-statement repeats its body as many times as there are lines in `file`. Since we're emphasizing mathematical structures throughout this book, iterations like this one — those that iterate over all elements of a sequence or set or other mathematical structure — will be especially important to us.

The general form of a `for`-statement is this:

```
for name in expression :
    statements
```

The expression provides the values to be iterated over. To give a simple example, the expression's value can be a string, which is a sequence of characters, as in the following:

```
sentence = "Look at me swimming!"

vowels = 0
for c in sentence:
    if c in "aeiou":
        vowels += 1

print("There are " + str(vowels)
      + " vowels in that sentence.")
```

One way or another, the expression in the header must provide a sequence of values. For each of those values, the Python interpreter binds the name in the header to the value and then executes the body of the for-statement.

Python for-statements can iterate over many kinds of things. Some of them are composite values that exist in advance, like strings. Others are sequences of values that are generated dynamically, one at a time, by objects called *iterators*. For example, look at this little program again:

```
file = open("names")
for line in file:
    if line.startswith("John"):
        print(line)
```

Here's what really happens: the value that the function `open` returns is an iterator. The for-statement uses the iterator to generate values: on each repetition, the iterator reads a line of the file and returns it to the for-statement, which binds that string to `line` and executes the body of the for-statement.

A “range” object is another kind of value that is useful in iterations. For an integer value n , a function call `range(n)` produces a range object that generates the sequence of integers from 0 to $n-1$, much as an iterator does, without having to create the entire sequence in advance. The most common use of range is in the header of a for-statement, like this:

```
for i in range(n):
```

This causes the body of the for-statement to be executed n times, with `i` taking on successive values from 0 to $n-1$, acting as a counter. Here's a simple example that displays the powers of 2 from 0 to 19:

```
for i in range(20):
    print("2 to the " + str(i)
          + " power is " + str(2**i))
```

We'll see many more examples of for-statements in later chapters. Especially, in the chapters that discuss mathematical structures in more detail, we'll see many examples of for-statements that iterate over elements of those structures.

Python has one more kind of compound statement for doing iteration: the *while-statement*. Its form is as follows:

```
while condition :
    statements
```

Its meaning is this: the body of the while-statement is executed over and over until the condition in the header becomes false. Specifically, to execute a while-statement, the Python interpreter first evaluates the condition. If it is false, control passes immediately to whatever follows the while-statement. Otherwise, the interpreter executes the body and then evaluates the condition again, and so on.

In some ways the while-statement is more basic than the for-statement: any iteration that can be done with a for-statement can be done with a while-statement, although not always as easily. Look at this example again:

```
for i in range(20):
    print("2 to the " + str(i)
          + " power is " + str(2**i))
```

Here's how we can do the same computation using a while-statement:

```
i = 0
while i < 20:
    print("2 to the " + str(i)
          + " power is " + str(2**i))
    i += 1
```

In the for-statement version, the iteration mechanism is all on one line. In the while-statement version, we need three lines: one to initialize the counter *i* to zero, one to increment the counter, and one (the while-statement header) to test the counter to determine when the iteration is done.

Here's another situation in which some programmers would use a while-statement: reading all lines of a file. Python has a function `readline` for reading one line of a file at a time. Actually, `readline` is a particular kind of a function called a “method”, which we'll discuss later; for now, let's just say that `readline` can be used like this:

```
line = file.readline()
```

Each time this statement is executed, `line` is assigned the next line in `file`, as a string. When there are no more lines in the file, `line` is assigned the empty string.

So, to do some computation on every line of a file, we can use a pattern of code like this:

```
line = file.readline()
while line != "":
    computation using line
    line = file.readline()
```

Again, we have an initialization step, the first call to `readline`; a test; and a step that advances the iteration for the next repetition of the computation, the second call to `readline`. But, as we have seen, we can do the job more easily with a for-statement:

```
for line in file:
    computation using line
```

So it's usually easier to use a for-statement than a while-statement, whenever the for-statement is an obvious fit for the computation that needs to be done; that is, whenever you can identify an obvious sequence or other collection of values to iterate over. But for some computations there really isn't any.

Here's an example. This piece of program calculates the square root of a number using an algorithm called “Newton's Method”. We assume that the number is in the variable `n` and that it is non-negative; we will calculate a close approximation to its positive square root and leave that approximation in the variable `root`. (The function `abs` gives the absolute value of an integer or a floating-point number.)

```
guess = 1.0
while abs(n - guess*guess) > .0001:
    guess = guess - (guess*guess - n)/(2*guess)
root = guess
```

If you have studied calculus, you might be able to see how Newton's Method works here and why `guess*guess` gets closer and closer to `n` so that the iteration eventually terminates. In any case, notice that there's no sequence here, other than the sequences of values that the variable `guess` gets on successive executions of the while-statement body.

Terms introduced in this chapter

statement	conditional
side effect	if-statement
simple statement	Boolean
compound statement	substring
header	nesting
body	iteration
keyword	for-statement
comment	iterator
flow of control	while-statement

Exercises

1. Write a program that will display a bar chart based on data from a file. The file is called “data”, and it contains non-negative integers, one per line.

The output of your program should look like the following (for a file containing 7, 15 and 11, as an example):

```
##### 7
##### 15
##### 11
```

2. Take your solution to the previous exercise and modify it so that it imposes a maximum length of the bars. If an integer from the file is greater than

25, your program will display only a bar of length 25. Here's how a bar chart might look now (with different data):

```
##### 7
##### 25
##### 283
##### 11
```

3. Now take your solution to the previous exercise and modify it again. If an integer from the file is greater than 25, your program will display a bar of length 25, but with an ellipsis (" / ") in the middle, like this:

```
##### 7
##### 25
##### / ##### 283
##### 11
```

4. Experiment with the square-root program of Section 3.3. Put a `print` expression-statement in the body of the `while`-statement to display the value of `guess`, and see how long it takes the iteration to converge for different values of `n` and different initial guesses. What happens if `n` is a negative number?
5. The first line of our square-root program is this:

```
guess = 1.0
```

What happens if you change it to this?

```
guess = 1
```

Try it, and explain what you see.

6. If you have studied calculus, try to explain how the square-root program works. (Hint: we are solving for x in the equation $x^2 = n$; that is, $x^2 - n = 0$, where n is a constant. The derivative of $x^2 - n$ is $2x$. Draw a picture.)

Chapter 4

Python functions

4.1. Function definitions

In this chapter we'll see how to define functions in Python and see some of the many things we can do with functions.

As we have seen, some Python functions compute values and some produce side effects. The latter kind are called “subroutines” or “procedures” in some programming languages, but in Python both kinds are called “functions” and are defined in the same way.

Here's an example of a *function definition*. As we have seen, Python already has an absolute-value function, but if it didn't we could define our own like this:

```
def abs(n):  
    if n >= 0:  
        return n  
    else:  
        return -n
```

A function definition is a compound statement. Its header contains the function name and the name of the function's *parameter*. When the statement is executed, Python binds the function name to the code in the statement's body, as well as to other details such as the name of the parameter. The body is not executed yet. When the the function is called, the parameter name is bound to the value of the argument that is passed to the function, and the body is executed. A *return-statement*, a statement containing `return` and possibly an expression, terminates the execution of the function body. It also causes the value of the expression, if there is one, to be returned from the function, and this value becomes the value of the function-call expression.

Notice that the type of the parameter is not declared anywhere. In fact, the argument passed to this function can be any value for which the operations

“>=” and unary “-” are defined. Thus, the function is automatically overloaded for integer and floating-point arguments; we don't need to do anything special to make that happen.

As another example, here's how we could take the square-root computation from the previous chapter and package it as a function:

```
def sqrt(n):
    guess = 1.0
    while abs(n - guess*guess) > .0001:
        guess = guess - (guess*guess - n)/(2*guess)
    return guess
```

Until now, all the functions that we have seen have had exactly one parameter, but a function can have more than one. For example, the iteration in the function above terminates when `guess` is close to the true square root of `n`. The stopping criterion is that the square of `guess` equals `n` to within a tolerance of `.0001`, but we could define the function so that the tolerance is another parameter, like this:

```
def sqrt2(n,tolerance):
    guess = 1.0
    while abs(n - guess*guess) > tolerance:
        guess = guess - (guess*guess - n)/(2*guess)
    return guess
```

Then we would call `sqrt2` with two arguments. For example, the call `sqrt2(3, .00005)` would do the computation using `.00005` as the tolerance.

By the way, we saw another syntax for function calls in the examples of Chapter 1 and Section 3.3:

```
line.startswith("John")
line.split(",")
line = file.readline()
```

This is the syntax that Python uses for calls to a particular kind of function, called a “method”, which we will see in Chapter 11. For now, it's enough to say that the value before the “.” is used as the first argument in the function call, and any remaining arguments are enclosed in parentheses after the function name as usual.

Now what about functions that don't return values? One typical use for them is to produce output, perhaps as the result of a computation. For example, consider the script from Chapter 1 that calculates and prints the average of numbers in a file (Example 1.3). That script could be made into a function, perhaps like this:

```
def printAverageOfFile(fileName):
    sum = 0
    count = 0

    file = open(fileName)
    for line in file:
        n = int(line)
        sum += n
        count += 1

    print("Average of numbers in " + fileName + ":")
    print(sum/count)
```

Here, the name of the file is a parameter of the function. This design lets us easily perform the same computation on more than one file, like this:

```
printAverageOfFile("observations-April")
printAverageOfFile("observations-May")
printAverageOfFile("observations-June")
```

It is possible to define a Python function that produces side effects and then returns a value as well. However, many programming experts believe that a program's design is cleaner if every function has only one purpose, and either returns a value or causes side effects but not both. We'll usually stick to that principle in this book.

4.2. Recursive functions

As in most modern programming languages, a function definition in Python can be *recursive*, meaning that the function body can call the same function that is being defined. In other words, the function is defined at least partially in terms of itself.

The classic example is the factorial function. For a positive integer n , the factorial of n , written $n!$, can be defined mathematically this way:

$$n! = n \cdot (n-1) \cdot \dots \cdot 2 \cdot 1$$

This recursive definition has a different structure but has the same effect:

$$\begin{aligned} n! &= 1 \text{ if } n = 1 \\ n! &= n \cdot (n-1)! \text{ otherwise} \end{aligned}$$

Here is the corresponding Python function definition:

```
def factorial(n):
    if n == 1:
        return 1
    else:
        return n * factorial(n-1)
```

Then suppose `factorial` is called, say with an argument of 5. Notice how `factorial` will keep calling itself, first with an argument of 4, then 3, then 2, and finally 1. At that point `factorial` will immediately return 1. Then all the recursive calls to `factorial` will return their results in reverse order: `factorial(2)` returning $2 \cdot 1 = 2$, `factorial(3)` returning $3 \cdot 2 = 6$, `factorial(4)` returning $4 \cdot 6 = 24$, and finally `factorial(5)` returning $5 \cdot 24 = 120$.

A good recursive function definition has the following characteristics:

- It is defined by cases (in Python, usually with an if-statement). At least one of the cases is *not* recursive: it defines the result directly, rather than in terms of a recursive reference to the function being defined. Such a case is called a *basis case*. In the factorial example, there is one basis case, the case in which $n = 1$.
- Every sequence of recursive calls eventually leads to a call in which one of the basis cases applies. Usually, the basis cases treat the situations in which the argument is “small” or “simple” or “trivial” in some sense, and the recursive cases (the cases that are not basis cases) take the function's parameter and recursively call the function with an argument that is “smaller” or “simpler” or closer to the basis case in some way. In the factorial example, the recursive case takes the parameter n and recursively calls the function with the argument $n-1$. It is easy to see that every sequence

of recursive calls will eventually end with $n = 1$, assuming that n is a positive integer in the original call.

We won't give a comprehensive treatment of recursive programming in this book, but we will see recursive functions again several times. In any case, recursion can be a useful and powerful programming technique and you will probably have many occasions to use it throughout your programming career.

4.3. Functions as values

In Python, functions are values, much as integers and strings are. A function-definition statement binds a name to the function's definition, and in Python this works very much like binding a name to a value in an assignment statement. Let's explore some of the implications of these facts.

First, once we have defined a function, we can bind another name to that function. One way to do this is with an assignment statement, as in this example:

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

sq = square
```

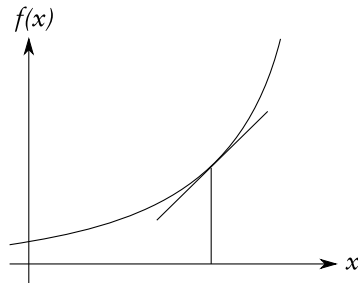
Now `sq` is bound to the same function definition that `square` is, so it computes the same thing. For example, the value of `sq(3)` would be 9.

We can also pass a function as an argument to another function. Here's an example: the function below takes as parameters a function and another value, and returns the result of applying the function to the argument twice.

```
def twice(f,a):
    return f(f(a))
```

So, for example, consider the function call `twice(square,3)`, where `square` is as defined above. The Python interpreter binds `f` to the function `square` and `a` to 3, and so returns `square(square(3))`, which is 3 to the fourth power, or 81. Similarly, the value of `twice(sqrt,3)` is the fourth root of 3.

Here's a somewhat more useful example. We can define a Python function that will compute an approximation to the derivative of a function at a given point. (For readers who haven't studied calculus: the derivative of a function is the rate of change of the function's value. Consider a function of one argument, $f(x)$. Picture a plot of $f(x)$ in the vertical direction against x in the horizontal direction. The derivative of the function at a given value of x is the slope of the curve at that value of x .)



The function below does the job. It takes a function `f`, a value `x`, and a small value `dx` to be used as the magnitude of a small interval. Then the function `derivAt` returns the change in value of `f` over an interval of length `dx`, starting at `x`, divided by the length of the interval, giving an approximation to the mathematical derivative of `f` at `x`.

```
def derivAt(f,x,dx):  
    return (f(x+dx) - f(x))/dx
```

Mathematically, the derivative of x^2 is $2x$, so the exact value of the derivative of the square function at the point 3 should be 6. Using the definition of `derivAt` above, `derivAt(square,3,.00001)` should give us an approximation to that value, and in fact Python does give us a value close to 6.

In Python a function can even return another function as a value. A function can execute a function-definition statement, creating a new function and binding that function to a name, and then return the value bound to that name. Here's an example. The function `multiplyBy` takes a number as a parameter and returns a function that multiplies by that number.

```
def multiplyBy(n):  
    def times(m):  
        return m*n  
    return times
```

Remember, a function definition is an executable statement. Here the definition of `times` is executed when the body of `multiplyBy` is executed. The value that is used for `n` in the new function is the value that is bound to `n` at that time, which is the value of the argument that is passed to `multiplyBy`. Since `m` is a parameter of the new function, it is not bound until that function is called.

We can use `multiplyBy` like this:

```
double = multiplyBy(2)  
triple = multiplyBy(3)
```

Then when we evaluate `double(7)`, `m` is bound to 7, the multiplication is done, and we get 14. Similarly, the value of `triple(33)` is 99.

The programming technique that we just used is a special case of a more general technique: to bind some but not all arguments of a function, leaving it a function of fewer arguments. This is called *partial application*. Here's a function that binds the first argument of any two-argument function, leaving it a function of one argument.

```
def partial(f,x):  
    def f_x(y):  
        return f(x,y)  
    return f_x
```

The function `partial` takes a two-argument function $f(x,y)$ and a value for x , and returns a one-argument function that we can denote as $f_x(y)$: it is $f(x,y)$ with the value of x held constant, and there is a different such function for each different value of x . For example, if `mult` is a function of two arguments that returns their product, the value of `partial(mult,2)` is the same function as `multiplyBy(2)`.

As a final example, let's define a function that will take the derivative of another function. It doesn't take the derivative symbolically, as in calculus,

but it returns a function that gives an approximation to the derivative at any given point. Here's how we can write it.

```
def deriv(f,dx):  
    def d(x):  
        return derivAt(f,x,dx)  
    return d
```

We call `deriv` with a function of one argument, say `square`, and a number to use as an interval size. The new function `d` uses those values for `f` and `dx` in its call to `derivAt`, but the new function is still a function of one argument, `x`. We can use `deriv` like this:

```
derivOfSquare = deriv(square, .00001)
```

Then the value of `derivOfSquare(3)` is a number close to 6, just as we found in the example using `derivAt`.

A function that takes another function as an argument, or returns a function, or both, is called a *higher-order function*. Our `twice` and `derivAt` and `partial` and `deriv` are all higher-order functions.

4.4. Lambda expressions

If functions are values in Python, are there expressions that have functions as their values? Yes, and they are called *lambda expressions*. The word “lambda” is the name of the Greek letter λ , and the term comes from mathematics.

In mathematics, a function that adds one to its argument can be denoted this way:

$$\lambda x . x+1$$

We can read that expression as “the function of x that is $x+1$ ”. The function doesn't have a name; it just exists.

That lambda expression would be written this way in Python:

```
lambda x: x+1
```

The value produced by a lambda expression can be bound to a name, passed to a function, or returned by a function, just as a function defined using `def` can. For example, consider the `square` function from the previous section:

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

We could have defined that same function this way:

```
square = lambda x: x*x
```

That is to say, `square` is the function of `x` that is `x*x`. For all practical purposes, the two definitions have the same effect.

Lambda expressions are handy for creating functions for other functions to return. For example, here's a version of `deriv` that is equivalent to the version in the previous section, but more concise:

```
def deriv(f,dx):  
    return lambda x: derivAt(f,x,dx)
```

Lambda expressions are also handy for creating functions to pass to other functions. For example, to take the derivative of the square-function, we can do it like this if we don't have the function `square` already defined:

```
derivOfSquare = deriv(lambda x: x*x, .00001)
```

In mathematics we sometimes use the vocabulary and notation of binary operators and two-argument functions interchangeably. For example, we may say that f is associative if $f(f(x,y),z) = f(x,f(y,z))$. We may even write $x f y$ instead of $f(x,y)$, or $*(x,y)$ instead of $x * y$. (Be warned: we'll sometimes take such liberties with notation in mathematical discussions in the chapters that follow.) But we can't interchange the notations as freely in Python. For example, we can't pass an operator like `*` to a higher-order function directly. The way to do it in Python is to pass an equivalent function such as `lambda x,y: x*y` instead.

We'll see other uses for lambda expressions in later chapters.

Terms introduced in this chapter

function definition
parameter
return-statement
recursive function

basis case
partial application
higher-order function
lambda expression

Exercises

1. For readers who know some calculus: experiment with the `deriv` function defined in this chapter and with the derivative functions that you get from `deriv`; try using `deriv` on `square` and other functions whose derivatives you know. Especially, experiment with increasingly smaller values of `dx`; let those values get very small. Do the results keep getting closer to the mathematically correct values of the derivatives at given points? Explain what you see.
2. What happens when a function definition contains a name that isn't bound to anything in the function definition?

```
def f(n):  
    return n + a
```

This is clearly an error if `a` isn't given a value anywhere.

```
a = 1  
def f(n):  
    return n + a
```

Now `f` is a function that adds 1 to its argument. But what about this case?

```
a = 1  
def f(n):  
    return n + a  
a = 3
```

After that, does `f` add 1 or 3? If you need to, experiment with the Python interpreter — can you explain what you see? Look again at the first page of this chapter.

Chapter 5

Tuples

5.1. Ordered pairs and n -tuples

In this chapter we'll start to explore mathematical structures and how they can be used in programming. We saw some of these mathematical structures in the examples of Chapter 1. Let's start with the *ordered pair*.

Here, “pair” simply means a grouping of two things. “Ordered” means that it makes a difference which is first and which is second. They may be different kinds of things, or they may have different meanings, or the distinction may be important in some other way. So an ordered pair is not the same as a set of two elements.

One important use for ordered pairs is to locate a place or point in two dimensions: latitude and longitude on the Earth's surface, for example, or horizontal and vertical coordinates in a plane. Complex numbers are similar: their real and imaginary parts are often pictured as coordinates in the “complex plane”.

An “ordered triple” is a similar mathematical object with three components instead of two. For example, to locate a point in three dimensions we need three coordinates, so they would be an ordered triple of numbers. We can omit the word “ordered” in situations in which it would be assumed, so we can say “triple” instead of “ordered triple” and even “pair” instead of “ordered pair”.

Similarly, an ordered grouping of four elements is called an “ordered quadruple” or just a “quadruple”. For more elements the names follow the same pattern: quintuple, sextuple, septuple, octuple, and so on.

But using Latin prefixes becomes a bit clumsy for large numbers of elements. Besides, we need a collective name for “tuples” of any size. The mathematical

term that we'll use is *n-tuple*. We'll substitute a number for *n* when we have a particular *n* in mind. So, for example, a 4-tuple is the same as a quadruple and a 2-tuple is the same as an ordered pair.

In mathematics, an *n-tuple* is typically denoted by components separated by commas and enclosed in parentheses, as in (3, 4) and (*x*, *y*, *z*). *N-tuples* are often used to construct complex mathematical objects out of more basic ones, as in the following typical definitions from mathematics and computer science:

A graph *G* is a pair (*V*, *E*), where *V* is a set of vertices and *E* is a set of edges.

A grammar is a 4-tuple (*N*, *T*, *P*, *S*), where *N* is a set of symbols called the *nonterminal alphabet*, *T* is a set of symbols called the *terminal alphabet*, *P* is a set of *productions* consisting of...

As the concept of an *n-tuple* is typically used in mathematics, an *n-tuple* contains a finite number of components and most often a rather small number. The components are not necessarily all the same kind of mathematical object. On the other hand, the components are not necessarily all different, as they are in a set.

5.2. Tuples in Python

Mathematical *n-tuples* can be represented in Python by the type called “tuple”. A Python tuple is created by using a comma as a binary operator, like this:

```
pair = 3, 4
quadruple = "Story number", 3, "is", True
```

The components of a Python tuple need not all have the same type, as the second example illustrates.

Python programmers often enclose tuples in parentheses, like this:

```
pair = (3, 4)
quadruple = ("Story number", 3, "is", True)
```

The parentheses are not strictly part of the tuple notation in Python, but they are syntactically necessary in many contexts. For example, to pass the tuple `3, 4` to the function `f`, one would write `f((3, 4))` because `f(3, 4)` would pass `3` and `4` as separate arguments. Furthermore, parenthesizing is consistent with mathematical notation, and (as we will see) it is symmetric with notation for other Python types such as lists and sets. Therefore, we will usually enclose our tuples in parentheses hereafter in this book.

There are two special cases. First, the empty tuple is written `()`. Second, a tuple with only one component is written as a single value followed by a comma, usually enclosed in parentheses like any other tuple. The comma is necessary to distinguish a one-component tuple from a single value: for example, `(3,)` is a tuple but `(3)` is just an integer expression whose value is `3`. Like the empty string, the empty tuple and one-component tuples may seem trivial but have their uses, as we will see.

The comma operator “packs” values together into a single object. To unpack a tuple, we can use a syntax that mirrors the syntax used to pack a tuple: an assignment statement with more than one name on the left-hand side, separated by commas. For example, if `pair` has the value `(3, 4)`, the following statement unpacks `pair` and gives `x` the value `3` and `y` the value `4`.

```
x, y = pair
```

We can also enclose the sequence of names on the left-hand side in parentheses, if we want to emphasize that we are doing tuple unpacking:

```
(x, y) = pair
```

This form might make it more clear that we are not simply binding both `x` and `y` to the whole tuple.

Here is another common use for the tuple-unpacking syntax: the simultaneous assignment.

```
x, y = y, x
```

This statement evaluates `y` and `x`, makes a tuple of them, and then unpacks the tuple and binds the values to `x` and `y`. The effect is as if Python assigns `y`

to x and x to y simultaneously. Here, the effect is to interchange the values of x and y without using a temporary variable to hold one or the other.

We can also extract a component of a tuple by position, numbering the components from 0. For example, if x has the value $(7, 13)$, the value of $x[0]$ is 7 and the value of $x[1]$ is 13. This syntax corresponds to the subscript notation in mathematics, in which those expressions would be written as x_0 and x_1 respectively.

5.3. Files and databases

Many applications of n -tuples in computing deal with collections of n -tuples of the same kind: that is, the n -tuples in a collection have the same number of components, and the corresponding components have the same type and meaning in each n -tuple. We saw an example in Chapter 1 (Example 1.2): the file that contained ordered pairs, with each pair containing a name and an email address. As we saw in that chapter, we often view the whole collection as a set or sequence of tuples.

We can view many computer files as collections of this kind. Typically, such a file is divided into lines, and each line is divided into fields. Each field may be a certain number of characters in length; or, more commonly nowadays, the fields may be separated by *delimiters*, such as the comma in the name-and-emails file in Example 1.2. Other common choices for the field delimiter are a tab character and a sequence of space characters.

Such files are often called *flat files*, “flat” meaning that the file has no structure other than being divided into lines and fields. When the delimiter is a comma, a flat file is said to be in *CSV format*, where “CSV” means “comma-separated values”. Many popular programs store their data in CSV format or can export their data in that format for other programs to import.

Many databases are essentially collections of n -tuples of data elements. In fact, the style of database called a *relational database* is explicitly defined in mathematical terms. A relational database is a set of relations, and each relation is a set of “tuples” that resemble our mathematical n -tuples. In Chapter 1 we

defined a relation as a set of ordered pairs, but such a relation is really just a special case called a *binary relation*. In mathematics, a relation is defined more generally as a set of n -tuples for some given n . A relation in a relational database is like that.

A database relation can be pictured as a table, and in fact the word “table” is often used interchangeably with “relation”. Each element of such a table is called a “tuple” or a “record”, and each component of a tuple is called a “field”. Fields are identified by name; in a picture of a relation, these names are shown as headings on columns of the table.

name	project	lab
Lambert	Alpha	221
Torres	Alpha	244
Malone	Beta	152
Harris	Beta	152
Torres	Beta	152

Strictly speaking, a tuple in a relational database is *not* the same as a mathematical n -tuple, because fields in a database tuple are identified only by name, not by position. This means that when we picture a relation as a table, the order of the columns is actually arbitrary. So a more accurate description of a database tuple is as a set of named fields, or as a mapping from field names to values.

In some simple relational database systems, the software stores the relations in flat files, one file per relation, and many database systems can at least export data in that form. A typical format for such files is a variant of CSV format, with some arbitrary order chosen for the fields in the tuples. The first line of the file contains the field names, separated by commas, and the remaining lines contain the tuples. Such a file is really not quite “flat”, since it does have this minimal structure. We can view such a file as an ordered pair whose first component is the line containing the sequence of field names and whose second component is a sequence, the lines representing the set of tuples.

```
project,name,lab  
Alpha,Lambert,221  
Alpha,Torres,244  
Beta,Malone,152  
Beta,Harris,152  
Beta,Torres,152
```

We will return to flat files, CSV files, and relational databases in later chapters.

Terms introduced in this chapter

triple, quadruple, etc.
 n -tuple
delimiter
flat file

CSV format
relational database
binary relation

Chapter 6

Sequences

6.1. Properties of sequences

We see the concept of “sequence” over and over in computing. We have seen examples of several kinds of sequences already:

- Strings, which are sequences of characters.
- Files, which (at least as seen by Python programs) are sequences of lines. Each line is itself a sequence of characters.
- The sequences generated dynamically by such Python objects as opened-file iterators and `range`.

Let's consider some of the properties that these and other sequences have in common.

First, notice that all these examples of sequences are not just generic sequences, but sequences *of* some particular kind of thing. Most of the sequences that we use in computing are homogeneous, meaning that all elements of the sequence are of the same kind, or at least we can view the elements that way by identifying a sufficiently general kind of thing that encompasses them all. For example, a line of a file may contain digits and letters and other kinds of characters, but we can view it as simply a sequence of characters.

So, as we usually use the terms, “ n -tuple” and “sequence” are somewhat different concepts, even though both n -tuples and sequences are ordered collections. We expect a sequence, but not an n -tuple, to be homogeneous. On the other hand, we expect a particular kind of n -tuple to have a fixed number of components. This is not necessarily true of sequences: for example, a file can contain any number of lines, from zero to many.

Now let's consider some mathematical properties of sequences. We'll use strings as an example, and then see how other sequences share some of their essential properties.

Recall that one important operation on strings is concatenation. The concatenation of two strings A and B is the string formed by taking the characters of A in order followed by the characters of B in order.

We can define concatenation of other sequences in exactly the same way. For example, the concatenation of two files is the lines of the first followed by the lines of the second, producing a third sequence of lines.

Concatenation of two sequences makes sense only if the sequences have the same kinds of components, so that the resulting sequence is homogeneous. For example, it doesn't make sense to concatenate a sequence of strings with a sequence of integers. But when two sequences do have the same kinds of components, the result of the concatenation is a sequence whose components are all of that same kind.

We have noted that there is an empty string, and this is a special case of the empty sequence, which is the sequence whose length is zero. Let us denote the empty string or sequence as e : then, in either case, e has the property that $e + a = a$ and $a + e = a$ for any string or sequence a , where “+” denotes concatenation. In mathematical vocabulary, e is an *identity* for concatenation.

Concatenation of strings has the associative property: if $+$ is the concatenation operator, $(a + b) + c = a + (b + c)$ for any strings a , b and c . In fact, exactly the same is true for concatenation of sequences of any kind.

However, concatenation of strings or other sequences is not commutative: $a + b \neq b + a$, unless a and b are the same or one of them is empty.

So let's summarize. Consider the set of all sequences of elements of a particular kind; call that set A . Then we have the following facts:

- The concatenation operation $+$ takes two operands from A and produces another member of A .

- Concatenation is associative, but not commutative.
- A contains an element e , the empty sequence, which is an identity for concatenation.

6.2. Monoids

It may not be just a coincidence that Python uses “+” as the concatenation operator for strings and (as we will see) for other kinds of sequences. The usage seems rather appropriate because concatenation is sometimes described informally as “adding” one sequence to the end of another. But concatenation is like adding in other ways. Consider the mathematical integers and the addition operation. The result of adding any two integers is another integer. Addition is associative. And addition has an identity: it is 0, because $0 + n = n + 0 = n$ for any integer n . So, to this extent, strings and concatenation behave like integers and addition.

Both strings with concatenation and integers with addition are examples of the mathematical structure called a *monoid*. A monoid is a set that has an associative binary operator and an identity element.

More formally, a monoid is an ordered pair (S, \otimes) such that S is a set and \otimes is a binary operator, satisfying these conditions:

1. For all a and b in S , $a \otimes b$ is defined and is also in S .
2. For all a , b and c in S , $(a \otimes b) \otimes c = a \otimes (b \otimes c)$.
3. There is an element e in S such that, for all a in S , $e \otimes a = a \otimes e = a$.

Then we also say that S is a monoid *under* \otimes , with identity e .

The concept of “monoid” comes from the field of advanced mathematics called “abstract algebra”. The name “monoid” may sound exotic, but the concept is simple enough, and there are many examples of monoids in mathematics and in programming. Here are the examples that we've already mentioned and several more.

- The mathematical integers are a monoid under addition, with identity 0. They are also a monoid under multiplication, with identity 1. Both operators are also commutative. Integers in Python have these same properties.
- The mathematical real numbers are a monoid under addition, with identity 0. They are also a monoid under multiplication, with identity 1. Both operators are commutative.

The Python floating-point numbers are not quite a monoid under addition: for floating-point operands, $(a + b) + c$ is often not exactly equal to $a + (b + c)$ because of roundoff error. The same is true of multiplication. But in many applications the equalities are close enough for computational purposes.

- Python Booleans are a monoid under `and`, with identity `True`. They are also a monoid under `or`, with identity `False`.

Notice that these claims are true even given the way Python evaluates those Boolean operators, sometimes giving a valid result even if the second operand is undefined or otherwise does not produce a Boolean value. Conditions 1–3 in the definition of “monoid” say nothing about the behavior of \otimes if either of its operands is not in the set S .

- Suppose that *max* is the maximum-function, so that $\text{max}(x,y)$ is defined to be x if $x \geq y$ and y otherwise. Recall that we can speak of *max* as if it were an operator and write $x \text{ max } y$ instead of $\text{max}(x,y)$. Then *max* is both associative and commutative, and the non-negative integers are a monoid under *max*, with identity 0.
- Consider the minimum-function *min* defined similarly. Then *min* is both associative and commutative, and the Python floating-point numbers are a monoid under *min*, with identity the infinite value obtained from `float("inf")`.
- Sequences of elements of a given set are a monoid under concatenation, with the empty sequence as identity.

- Python strings are a monoid under `+`, with identity `""`.

Later we'll see ways that we can take advantage of the similarities among these and other monoids.

A set that has an associative binary operator, but not necessarily an identity element, is called a *semigroup* in abstract algebra. A semigroup is like a monoid except that condition 3 is dropped from the definition. Every monoid is a semigroup, but not every semigroup is a monoid.

An example of a semigroup that is not a monoid is the set of positive integers under addition: 0 is not in the set of positive integers and so the set has no identity. We'll see a few more examples of semigroups without identities as we go along, but most semigroups that we'll talk about have identity elements and so are also monoids.

By the way, a *group* is a semigroup that has inverses as well as an identity. In other words, it is a monoid in which every element a has an inverse a^{-1} such that $a \otimes a^{-1} = e$ and $a^{-1} \otimes a = e$, where e is the identity. For example, integers under addition are a group, where the inverse of each number is its negative. Groups are very important in mathematics and some other fields, such as particle physics, but most of the monoids that are important in programming are not groups. For example, strings under concatenation are not a group, because most strings don't have inverses. There is no string that you can concatenate to a nonempty string to get an empty string.

Some monoids have properties other than properties 1–3 in the definition of a monoid. For example, consider the mathematical integers under addition; they are not only a monoid but also a group. Besides, addition is not only associative but also commutative. Integers also have multiplication, and the two operators are related by the distributive property: $a(b + c) = ab + ac$. Integers are ordered by the “less than” relation, each integer has a unique prime factorization, and the integers have many more properties. A mathematician would say that the integers have more “structure” than that implied by the fact that they are a monoid under addition.

Sequences, on the other hand, have *no* such additional structure. For a given set S , the set of all finite sequences of zero or more elements of S forms (under concatenation) what mathematicians call a “free monoid”, “free” in the sense that the monoid obeys no laws or constraints other than those implied by the definition of a monoid. This means that, in programming, such data types as strings and lists are blank slates: they can be used to represent and store almost any kind of data that one can imagine.

Strings under concatenation are a classic exemplar of a monoid. To say that a set and an operation form a monoid, then, is to say that the set and operation behave, at least to some extent, like strings and concatenation. They may have more properties and behavior, but they have at least the properties and behavior of strings and concatenation. We can think of strings to remember how monoids behave. If we prefer, we can even say “string-like object” instead of “monoid” or “is string-like” instead of “is a monoid”, although the term “monoid” is concise and expresses the concept exactly.

We can often transfer what we know about strings and concatenation to other monoids in programs, translating vocabulary appropriately. For example, let's consider an expression that contains several instances of the string concatenation operator:

$$s_0 + s_1 + s_2 + \dots + s_n$$

Because the operator $+$ is associative, we don't need to put parentheses in the expression to indicate which concatenations are done first, and we don't need rules to say that the evaluation is left-to-right or right-to-left or in any other order. We don't need to think of evaluation order at all; we can look at the expression as simply a sequence of operands that are all combined with the $+$ operator.

This line of thinking leads us to the idea of a “big $+$ ” operator that, instead of taking two operands, takes a sequence of operands and concatenates them all:

$$\begin{array}{cccccc}
 s_0 & s_1 & s_2 & \dots & s_n \\
 \underbrace{\hspace{10em}} \\
 +
 \end{array}$$

It's not hard to see how we could write Python code to implement the “big +” operation. We leave the details as an exercise.

Once we know how “big +” works and how to implement “big +” for strings, we can transfer what we know to any other monoid. For any monoid operator \otimes , we can define a “big \otimes ” operator that works in much the same way as “big +”:

$$\begin{array}{cccccc}
 s_0 & s_1 & s_2 & \dots & s_n \\
 \underbrace{\hspace{10em}} \\
 \otimes
 \end{array}$$

And whatever \otimes may be, we can often use any Python code that implements or uses “big +” as a model for Python code that implements or uses “big \otimes ”.

The “big” operators (as we have called them) are common in mathematics, and there are traditional symbols for some of them. For example, the “big +” for integers and real numbers is Σ (the uppercase Greek letter sigma, for “sum”), and the “big *” for numbers is Π (the uppercase Greek letter pi, for “product”).

Everything that we have said about monoids has implications for programmers, and we'll see some of these implications in the sections and chapters that follow.

6.3. Sequences in Python

We have seen one Python construct for representing sequences, the tuple; another is the type called “list”. In fact, a Python tuple can represent either a mathematical n -tuple or a sequence, and so can a Python list.

Lists are created by separating a sequence of values by commas and enclosing the whole sequence in square brackets, like this:

```
[2, 3, 5, 7, 11]
```

This construct is called a *list display*. The empty list is denoted by the list display `[]`. A list display with only one component can be written with just square brackets but no comma, since that syntax has no other possible meaning: an example is `["John"]`.

We may think of a construct like `(2, 3, 5, 7, 11)` as a “tuple display”, but it isn't really. As we have seen, in this context the comma is a binary operator, and it is the commas and not the parentheses that produce the tuple. But the tuple notation using parentheses and the notation of list displays using square brackets are very similar and ultimately have similar meanings, a fortunate consequence of Python's syntax rules.

Lists and tuples are alike in many ways. For example, positions in a list are numbered from 0, and a component can be selected using the subscript notation. If the name `a` is bound to the list `[2, 3, 5, 7, 11]`, then `a[2]` is 5.

Lists, like tuples, can be unpacked using an assignment statement with more than one name on the left-hand side. That's what we did in the script of Example 1.2:

```
name, email = line.split(",")
```

The value returned by the function `split` is a list, which has two components if `line` contains a single comma. The assignment statement unpacks this list into two variables.

The difference between tuples and lists is that lists are *mutable objects*: their values can be changed in place. For example, the subscript notation can be used on the left-hand side of an assignment statement to change a component of a list. Consider these statements:

```
a = [2, 3, 5, 7, 11]
a[2] = False
```

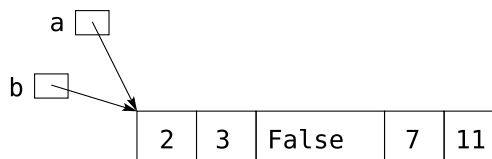
After these statements are executed, the name `a` is still bound to the same object, the list, but part of that object has been altered: the object's value is now `[2, 3, False, 7, 11]`. (Notice that lists can be heterogeneous just as tuples can be.)

Now that we have mutable objects, we need to observe the distinction between objects and their values. In Python, names are bound to objects, not directly to values. Suppose we do this:

```
a = [2, 3, 5, 7, 11]
b = a
a[2] = False
```

The assignment `b = a` binds `b` to the same object that `a` is bound to; in other words, it creates an *alias* for that object. We must be careful when we create aliases for mutable objects, because the effect may not be what we want. In this case, `a` is bound to an object that now has the value `[2, 3, False, 7, 11]`, but so is `b`, even though we have done nothing explicit to change `b`.

If you are familiar with the concept of a pointer from other programming languages, you can think of the Python bindings in this example as like pointers to objects. The first line creates a list and causes the value of `a` to be a pointer to that list. The second line copies the pointer to `b`, so that now there are two pointers to the same list.



Python has a generalization of subscripting called “slicing”: if `x` is a tuple, `x[i:j]` is another tuple containing the components of `x` at positions `i` up to but not including `j`, and similarly if `x` is a list. For example, after the statements above, `a[0:2]` is the list `[2, 3]`. One useful variation on this syntax is the slice with no upper bound: in this case, the slice continues to the end of the sequence. For example, `x[1:]` contains all components of `x` except the first.

Strings are sequences just as tuples are, so subscripting and slicing can be done on strings too. For example, if `name` is `"John Davis"`, then `name[0:4]` is the string `"John"`.

If `a` is a tuple or a list or a string, `len(a)` is the length of `a`; that is, how many components it has. In the case of a string, this is the number of characters in it.

Some other operations on sequences use the method-call syntax. We have already seen the `startswith` method for strings. Other methods can be applied to any Python sequence. An example is `count`, which counts the number of occurrences of a value in a tuple, list, or string; for example, if `a` has the value `"Look at me!"`, then `a.count("o")` has the value 2.

Python has methods for modifying a list in place; these methods can only be applied to lists, not tuples or strings, because only lists are mutable. An example is `append`, which appends a value to the end of a list. For example, suppose `b` has the value `[1, 2, 3]`. Then, after the statement `b.append(4)`, is executed, `b` has the value `[1, 2, 3, 4]`.

The `+` operator can be used to concatenate two tuples or two lists, and the result is another sequence of the same kind. So the set of all Python tuples is a monoid under `+`, and so is the set of all lists. So is the set of all tuples or lists of elements of a particular type or having some particular property, such as all lists of strings, or all tuples of names of European cities. (Python does not overload `+` for concatenation of a tuple and a list, or a list and a string, or any other such combination.)

So, except that lists are mutable, tuples and lists have almost identical behavior, and a programmer can choose either for many purposes in programs.

Generally, though, we will use tuples rather than lists hereafter in this book except where we need sequences that are mutable.

Like the type conversion function `int`, there are functions `tuple` and `list` to convert any other kind of sequence to a tuple or list respectively. For example, the value of `tuple([2,3,5,7,11])` is `(2,3,5,7,11)` and the value of `list("Hello")` is `["H","e","l","l","o"]`. We can even use `tuple` or `list` on an object that generates a sequence of values, such as a `range` object or an iterator. For example, the value of `tuple(range(5))` is `(0,1,2,3,4)`; here `tuple` forces `range` to generate all of its values at once to pack into the tuple.

A Python `for`-statement can iterate over the components of any kind of sequence. We have seen several examples already, but here is another using a tuple:

```
for place in ("London", "Paris", "Istanbul"):
    print("Hello from " + place + "!")
```

Iterating over a tuple of constants, as in this example, is a handy way to do the same operation for each of a fixed set of values.

6.4. Higher-order sequence functions

We can perform a wide range of computations on sequences easily and elegantly by using higher-order functions. Let's develop this idea by defining a classic trio of higher-order functions — `map`, `filter`, and `reduce` — and exploring some ways to use them.

The first, `map`, applies a one-argument function to every element of a sequence, producing a tuple of the results. A call to `map` will take this form:

```
map(f, sequence)
```

For example, applying `map` to functions that we defined in Section 4.3, `map(double, (2,3,5,7,11))` is `(4,6,10,14,22)` and `map(square, range(10))` is `(0,1,4,9,16,25,36,49,81)`.

Here's how we can define the `map` function. It's not hard:

```
def map(f, sequence):
    result = ( )
    for a in sequence:
        result += ( f(a), )
    return result
```

Are you concerned that the “`result +=...`” tries to modify `result` in place, when `result` is an immutable object? Don't worry. Recall that “`result +=...`” means “`result = result + ...`”, so we're not really modifying `result` in place.

Notice that `sequence` can be any value that generates a sequence when used in the header of the for-statement: it can be a tuple, a list, a `range`, or an iterator.

Python has a similar `map` function built in, but it doesn't work in quite this way. We'll see how it does work in the next chapter.

Our second function, `filter`, takes a one-argument function that returns a Boolean, and a sequence. The `filter` function applies the given function to each element of the sequence and returns a tuple of only the elements for which the function returns `True`. A call will take this form:

```
filter(test, sequence)
```

For example, `filter(lambda x:x>0, (2,3,0,-5,7,-11))` is `(2,3,7)`.

Again, Python has a similar function built in, but we can easily define our own version. We leave this task as an exercise.

Our third function, `reduce`, takes another function and a sequence of values, and reduces the values to a single value by combining them using the function. A call will take this form:

```
reduce(f, sequence, initial)
```

Here, `f` will be a two-argument function that acts like a binary operator, and `reduce` will use it to combine successive pairs of values as if the operator were placed between those values. The value `initial` will be used as a starting value; it will be the value returned as a default if the sequence is empty.

For example, suppose that `plus` is a two-argument function that does addition:

```
plus = lambda x,y: x+y
```

Then suppose that a is a sequence of integers a_0, a_1, a_2, \dots and i is some integer value. Then the value of `reduce(plus, a, i)` is

$$i + a_0 + a_1 + a_2 + \dots$$

In particular, the value of `reduce(plus, a, 0)` is simply the sum of the elements of a . To describe it another way, it is the “big +” operation of Section 6.2 applied to a .

By now it should be easy for us to implement `reduce`. In fact, the code is surprisingly similar to the code for `map`:

```
def reduce(f, sequence, initial):
    result = initial
    for a in sequence:
        result = f(result, a)
    return result
```

Now notice that we can use `reduce` on *any* values and operation that form a monoid, using the monoid identity as the starting value. In fact, `reduce` gives us an easy way to define any of the “big” operations that we described in Section 6.2.

Specifically, suppose that a set A is a monoid under \otimes , with identity e . Suppose that the function f implements the monoid operation, so that $f(x,y) = x \otimes y$. And suppose that S is a sequence of values a_0, a_1, a_2, \dots from A . Then

$$\begin{aligned} \text{reduce}(f, S, e) &= e \otimes a_0 \otimes a_1 \otimes a_2 \otimes \dots \\ &= a_0 \otimes a_1 \otimes a_2 \otimes \dots \end{aligned}$$

In other words, it is “big \otimes ” applied to S .

For example, if s is a sequence of numbers (integer or floating-point or any combination), the value of `reduce(plus, s, 0)` is the sum of the elements of s , as we have seen. Let's define that sum as a function:

```
sum = lambda S: reduce(plus, S, 0)
```

Then `sum` is a one-argument function that acts like “big +”. We can use the same pattern for the product of the elements of a sequence:

```
product = lambda S: reduce(lambda x,y: x*y, S, 1)
```

Or suppose `S` is a sequence of strings: the concatenation of all of the strings in `S` is

```
cat = lambda S: reduce(lambda x,y: x+y, S, "")
```

As a matter of fact, we could write that as `... reduce(plus, S, "")` since `plus` (as we defined it above) is automatically overloaded for strings and is concatenation when used that way.

Notice that our function definition for `reduce` causes it to perform its operation left-to-right:

$$\text{reduce}(f, S, e) = (\dots (((e \otimes a_0) \otimes a_1) \otimes a_2) \otimes \dots)$$

Knowing this, we can use that implementation of `reduce` with operations that are not necessarily associative, as long as left-to-right is the evaluation order that we want. But suppose we need right-to-left evaluation. Then we would want a function similar to `reduce` that acts this way:

$$\text{reduceRight}(f, S, e) = (a_0 \otimes (a_1 \otimes (a_2 \otimes (\dots \otimes e) \dots)))$$

One way to define `reduceRight` is recursively, like this:

```
def reduceRight(f, sequence, initial):
    if len(sequence) == 0:
        return initial
    else:
        return f(sequence[0],
                 reduceRight(f, sequence[1:],
                             initial))
```

Of course, if the operation is associative, as with a monoid, the evaluation order does not matter, and we can use `reduce` and `reduceRight` interchangeably.

We can do a surprising variety of programming tasks using little more than the `map`, `filter`, and `reduce` functions and other functions defined using them. For example, we can rewrite the sample scripts of Section 1.2 using only one Python expression apiece.

Here is the first of those scripts (Example 1.1) again. It displays every line of the file `names` that starts with “John”.

```
file = open("names")
for line in file:
    if line.startswith("John"):
        print(line)
```

Example 6.1 shows another way to write it. The iterator `open("names")` produces a sequence of strings, each ending in a “new line” character. We use `filter` to get only the strings that start with “John”; and we use `cat`, which we defined above using `reduce`, to combine those strings into a single string. Because of the “new line” characters, that string is broken into lines when we display it using `print`. If it weren't for this fortunate fact, we could break the lines properly anyway by using `map` to insert a “new line” character (“`\n`” in Python) after each name — can you see how?

Example 6.1. Finding a name again, in functional style

```
print(cat(filter(lambda x: x.startswith("John"),
                open("names"))))
```

We leave the task of rewriting the second script (Example 1.2) as an exercise; here is the third script (Example 1.3) again. It displays the average of the integers in the file `observations`.

```
sum = 0
count = 0

file = open("observations")
for line in file:
    n = int(line)
    sum += n
    count += 1

print(sum/count)
```

To do the equivalent computation, we use `map` with the conversion function `int` to convert the sequence of file lines to a tuple of integers. To that tuple, we apply a function that divides the sum of a tuple's elements (which we get using the function `sum` that we defined above using `reduce`) by the length of the tuple. Example 6.2 shows the complete program.

Example 6.2. Average of observations again, in functional style

```
print((lambda a: sum(a)/len(a))(
    map(int, open("observations"))))
```

The style of programming that we are using here is called *functional programming*. In functional programming, instead of writing sequences of statements, we write nestings of function calls. Instead of storing intermediate results in variables, we take the values returned by function calls and use them directly as arguments in other function calls. Instead of iterations, we use recursion and higher-order functions like `map`, `filter`, and `reduce`.

There are “functional programming languages” that emphasize functional programming almost to the exclusion of any other style of programming; some that you may encounter are LISP and its dialects such as Scheme, ML and its dialects, and Haskell. But functional programming is easy enough to do in Python, and it gives you a useful collection of techniques to add to your repertoire.

6.5. Comprehensions

Another Python construct for creating lists is the list *comprehension*, which is another kind of list display. A list comprehension constructs a list from one or more other sequences, applying function application and filtering in much the same way as the `map` and `filter` functions do, but with a syntax that perhaps suggests the structure of the result more directly.

Here is the most basic form of a list comprehension:

```
[ expression for name in sequence ]
```

This binds the *name* to successive values from the *sequence*, evaluates the *expression* with each such binding, and makes a list of the results. Here is an example:

```
[ 2**n for n in range(5) ]
```

The value of this expression is a list of the first five powers of 2, or `[1,2,4,8,16]`. In other words, it is the same as

```
list(map(lambda n: 2**n, range(5)))
```

A comprehension is somewhat more general than an application of `map`, because it can contain more than one “`for ... in`” phrase. Here is an example:

```
[ (a,b) for a in range(6) for b in range(5) ]
```

This produces a list of all thirty different ordered pairs of numbers whose first component is in the range `0...5` and whose second component is in the range `0...4`.

A comprehension can also filter the sequence that it draws values from. Then the syntax is

```
[ expression for name in sequence if test ]
```

This is much like `map(expression, filter(test, sequence))`, except that the *expression* and the *test* are expressions rather than functions and the result is a list.

Here is an example:

```
[ line[5:] for line in open("names")
  if line.startswith("John ") ]
```

This produces a list of strings, selecting the lines that start with “John ” in the file `names` and taking the remaining characters of each such line. In other words, the result is a list of the surnames of people named John, assuming that each name in the file is just a first name and a surname.

What about tuple comprehensions? We might think that we could write

```
( line[5:] for line in open("names")
  if line.startswith("John ") )
```

and get a tuple of the surnames instead of a list. Python has a surprise for us. Yes, this syntax is good Python, but instead of a tuple comprehension it is a *generator expression*, and its value is an iterator that generates a sequence dynamically. We'll see more about generator expressions in the next chapter; for now, let's just say that in many situations we can use a generator expression as we would use a tuple comprehension if Python had such a thing.

As we'll see in later chapters, Python also has comprehensions for sets and mappings.

6.6. Parallel processing

The world constantly demands computers that can process more data faster. Until recently computer makers have met the demand by improving traditional processors — those that perform a single operation on a single piece of data at a time — making them faster, smaller and cheaper every year. But this kind of progress cannot continue forever. Already chip makers are fighting against fundamental physical limitations such as the size of molecules, the speed of light, and the heat generated by running a chip at a high clock rate.

The future of computing clearly lies in *parallel processing*: performing many operations on many pieces of data at the same time. And the future is already upon us. Parallel-processing hardware once existed only in research labs and at supercomputer sites, but now many of us use it every day:

- Many of our desktop and laptop computers now contain multicore processors: those that contain two or more of the elements that we used to call “processors” within a single chip or group of closely-connected chips. Newer computers are being produced with more and more cores every year.
- Specialized graphics processing units (GPUs) are now inexpensive enough that they are built into many desktop and laptop computers, as well as other devices such as gaming consoles and even mobile phones. Current GPUs contain many specialized computation elements that operate in parallel, and manufacturers are producing GPUs with more and more parallelism every year.

Parallel computing will be one of the great challenges for programmers in the coming years, and programmers will need to learn new skills and think in new ways. Our traditional style of program is sequential, with a single flow of control through the code. How can we write code that produces the same results, but faster, using many computations executing at the same time?

It won't always be easy. Parallel computations must collaborate with each other, and one of the big difficulties is to keep them from interfering with each other instead. Some parts of computations can't safely be done at the same time because they may interfere with each other, creating an incorrect result. Think of two programs trying to write to a file at the same time, for example.

The computer science community has spent decades developing techniques for organizing and synchronizing parts of parallel computations so that they don't interfere with each other, and this technology is well beyond the scope of the current book. But let's examine the implications of one obvious principle:

Two computations can be done in parallel if they are independent of each other; in other words, if whatever one computation does can have no effect on what the other does.

For example, consider a sequence of assignment statements of the form

$$\begin{aligned}x &= E \\ y &= F\end{aligned}$$

If the expression F does not access x , and if E does not access y , and if the evaluation of E has no side effects on variables that occur in F and vice versa, then the assignment statements are independent of each other and can be executed in parallel. Otherwise, they are not independent and must be performed in sequence.

It is not always easy to analyze code to determine what parts of it are independent of one another, or to write code in which parts of it are guaranteed to be independent of one another. But here's some good news: in the world of functional programming, many parts of our computations are automatically independent of one another, so we get plenty of opportunities to do computation in parallel.

Let's consider computations — programs or parts of programs — that consist entirely of evaluating expressions and applying functions, in which no expression or function has side effects. In such computations, many opportunities for parallel computation are easy to identify:

- Consider a function call of the form $f(P, Q)$ or an expression of the form $P \otimes Q$. Since evaluating P is independent of evaluating Q , they can be evaluated in parallel. The same principle applies to function calls with three, four, or more arguments.
- Consider a map operation of the form $\text{map}(f, S)$. Then applying f to any element of S is independent of applying f to any other element. Therefore, *all* these computations can be done in parallel.
- The same is true of a filtering operation of the form $\text{filter}(test, S)$. The *test* can be done on all components of S independently, and thus all the tests can be done in parallel.

- Consider a reduce operation of the form $\text{reduce}(f, S, e)$, where f is associative and has identity e , as in a monoid. Suppose we break the sequence S into sub-sequences S_1 and S_2 , so that $S = S_1 + S_2$. Then, by associativity of f ,

$$\text{reduce}(f, S, e) = f(\text{reduce}(f, S_1, e), \text{reduce}(f, S_2, e))$$

Thus we can reduce S_1 and S_2 separately, and in fact in parallel, and then combine the results using f . In fact, we can break S_1 and S_2 into even smaller sequences and reduce them in parallel in the same way, and so on.

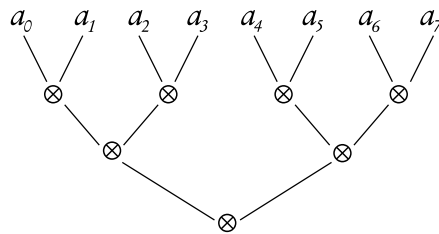
For example, suppose that S has eight components $a_0 \dots a_7$ and that f implements the monoid operation \otimes , so that the reduce operation computes the expression E below:

$$E = a_0 \otimes a_1 \otimes a_2 \otimes a_3 \otimes a_4 \otimes a_5 \otimes a_6 \otimes a_7$$

By associativity, E is equal to

$$((a_0 \otimes a_1) \otimes (a_2 \otimes a_3)) \otimes ((a_4 \otimes a_5) \otimes (a_6 \otimes a_7))$$

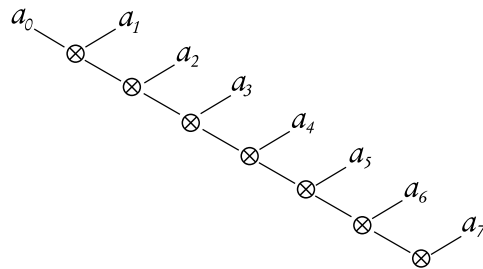
So to evaluate E we can apply \otimes to four pairs of components in parallel, then take those results and apply \otimes to two pairs of them in parallel, and finally apply \otimes to those two results.



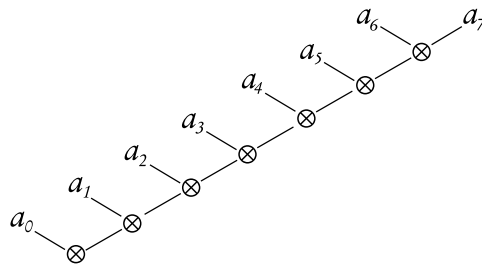
A diagram of this kind is called a *tree* in mathematics and computer science: here the “root” is at the bottom and the “leaves” are at the top. The tree shows each operator connected to its operands. Since \otimes is a binary operator,

each operator in the tree is connected to two operands, and so this is called a *binary tree*.

We can draw a similar tree to illustrate how E would be evaluated when evaluation is entirely left-to-right:



And with right-to-left evaluation, the tree would look like this:



But the first of these three trees is “balanced”. A balanced tree has the fewest possible levels for the number of leaves that it contains.

In fact, the number of levels of operators in a balanced binary tree with n leaves is $\log_2 n$, rounded up. In the first tree, for example, there are 8 operands and $\log_2 8 = 3$, so there are 3 levels of operators.

Let us say that we are using “balanced parallel evaluation” when we structure an expression evaluation as a balanced tree in this way and do all operations that are at the same level in parallel. Now let's define a time unit as the time it takes to perform one \otimes operation. Then if we can perform as many

computations in parallel as necessary, we can evaluate an expression of this kind most efficiently by using balanced parallel evaluation, and do the evaluation in $\log_2 n$ time units.

In that analysis we ignore all overhead — the time it takes to allocate computations to different processors, transmit the results back, coordinate all the computations, and so on — and in practice this overhead time can add significantly to the total time required. But if \otimes is a sufficiently time-consuming operation, taking a long time compared to the overhead time, balanced parallel evaluation can speed up the computation significantly.

You might imagine a version of the `reduce` operation that does its reduction in parallel as much as possible by using balanced parallel evaluation. That version of `reduce` would require its function argument to be associative, of course.

We won't discuss *how* to arrange for computations to be done in parallel, in Python or otherwise. Again, that topic is beyond the scope of this book, and is likely to depend heavily on the technology available in a particular computing environment at a particular time. But the main principle — that independent computations can be done in parallel — and the implications of that principle are general and broadly applicable and should be relevant in any particular case.

Yes, adjusting to the world of parallel computation will be a challenging task for programmers. But programmers who know how to think of data and operations in terms of their mathematical properties, and who know how to write computations using higher-order operations and functional programming, are likely to have a head start.

In the chapters that follow, we'll point out more opportunities to apply our mathematical thinking to parallel computing.

Terms introduced in this chapter

identity
monoid

alias
functional programming

semigroup
group
list display
mutable object

list comprehension
parallel processing
tree
binary tree

Exercises

1. In Python floating-point arithmetic, try to predict some values of a , b and c for which $(a + b) + c$ is not exactly equal to $a + (b + c)$. Explain your reasoning. Check your predictions with the Python interpreter.
2. Are percentages (the mathematical real numbers between 0 and 1 inclusive) a monoid under addition? Under multiplication?
3. Are Python integers a monoid under `**`? If so, what is the identity?
4. Are the Python `and` and `or` operators commutative?
5. The field of abstract algebra contains many interesting and useful results about semigroups and monoids. Here's one: a monoid can contain only one identity element. That is, if e_1 and e_2 are both identities in the same monoid, then it must be the case that $e_1 = e_2$.

Show why this is true. (Hint: consider $e_1 \otimes e_2$, where \otimes is the monoid operator.)

6. Define your own version of the function `filter` without using the built-in Python function with that name. Make your function return a tuple.
7. The functions `map` and `filter` are actually special cases of `reduce`. Define versions of `map` and `filter` using `reduce` instead of iteration or recursion.
8. Define a higher-order function `big` that takes a function and an identity element as arguments, and produces a one-argument function that acts like a "big" version (Section 6.2) of the given function. For example, if the function `plus` is defined as in Section 6.4, the value of `big(plus, 0)`

will be a function that takes a sequence (tuple, list, etc.) of numeric values and returns their sum.

9. Define a version of the function `reduce` that has only two parameters and omits the third, `initial`. Use the first element of the sequence as the starting element instead. Your function will be usable only on sequences of at least one element; if a sequence contains only one element, the function will return that element.

This is an appropriate reduction operation for a semigroup that is not a monoid and so does not have an identity element to use as an initial value.

10. Suppose that we have a function `reverse` that reverses a sequence, so that `reverse((1,2,3))` is `(3,2,1)`. A student suggests that we can define `reduceRight` simply by reversing the sequence and applying `reduce`, and reversing the result back, like this:

```
def reduceRight(f, sequence, initial):  
    return reduce(f, reverse(sequence), initial)
```

What is wrong with this solution? Under what circumstances will this version of `reverseRight` give an incorrect result?

11. Rewrite the second script of Section 1.2, the script that displays the email address of John Davis, in a single Python expression.

Chapter 7

Streams

7.1. Dynamically-generated sequences

Some sequences of data are static data structures, like the strings and tuples and lists of Python. A “static” sequence may be mutable, like a Python list, but at any one time it exists as a complete data structure. Some sequences, on the other hand, are generated dynamically.

An input stream, for example, appears to a program to be a sequence of values — lines, characters, numbers from sensors, whatever they may be — that are not present all at once, but appear dynamically over time. Some input streams don't even have an end: the data keeps coming indefinitely.

Let us use the term *stream* for any dynamically-generated sequence of values. So sequences are of two kinds: static sequences and streams.

In Python, streams are generated by two kinds of objects that we have seen: iterators and range objects. A range object is not quite the same as an iterator: there are a few more things that you can do with it. For example, you can apply the `len` function to a range object but not to an iterator. But, for our purposes, range objects are sufficiently similar to iterators that we will seldom need to distinguish between them. Except where we specifically mention that range objects are different, what we say about iterators hereafter will apply to range objects as well.

The most common way to invoke an iterator is with a for-statement:

```
for name in iterator :  
    statements
```

Here's a familiar example (Example 1.1); you'll remember that the `open` function returns an iterator.

```
file = open("names")
for line in file:
    if line.startswith("John"):
        print(line)
```

By the way, what happens when the value after the “in” isn't an iterator, but is a static sequence, as in this example?

```
for place in ("London", "Paris", "Istanbul"):
    print("Hello from " + place + "!")
```

Here's what really happens: Python constructs an iterator to generate values from the sequence, and the for-statement uses that iterator, invoking it over and over until it returns no more values.

As we saw in the previous chapter, Python has a construct — the generator expression — that looks like it would be a tuple comprehension but that instead generates a sequence dynamically. In other words, a generator expression generates a stream.

A generator expression has the same syntax as a list comprehension but with parentheses instead of square brackets on the outside, like this:

```
( line[5:] for line in open("names")
  if line.startswith("John ") )
```

The value of a generator expression is an iterator. As we have seen, an iterator is an object, and we can bind a name to it, pass it to a function, and so on.

This particular generator expression is interesting: it is defined using another iterator, the one returned by the `open` function. In fact, it is rather common for iterators to be defined using other iterators, as we will see. Watch what happens when we use this one.

```
surnames = (line[5:] for line in open("names")
            if line.startswith("John "))
for s in surnames:
    print(s)
```

The generator expression is evaluated, creating an iterator, and the name `surnames` is bound to that iterator. The for-statement invokes `surnames` for

values one at a time. Each time, `surnames` invokes the opened-file iterator to get another line, causing a line to be read from the file and bound to `line`; then `line[5:]` is computed and bound to `s`, and the body of the for-statement is executed.

By the way, what happens if we try to use the same iterator a second time?

```
surnames = (line[5:] for line in open("names")
            if line.startswith("John "))
for s in surnames:
    print(s)
for s in surnames:
    print(s)
```

The second time, the iterator generates nothing! When we take a value from an iterator, it is as if the value is “consumed”. A for-statement, since it iterates over all values that the iterator generates, leaves the iterator “empty”.

7.2. Generator functions

Python has one more construct for generating sequences dynamically: the *generator function*. This is a function that returns an iterator. We have already seen one such function: the built-in function `open` that opens a file and returns an iterator that generates strings from the lines of the file. Now we'll see how programmers can define their own generator functions.

A generator function yields a value by executing a *yield-statement*, which has this form:

`yield expression`

Python treats any function definition that contains a yield-statement as defining a generator function instead of an ordinary function. Here is an example of a generator function:

```
def fibs(n):
    a = 1
    b = 1
    for i in range(n):
        yield a
        a, b = b, a + b
```

The function `fibs` generates a sequence of length n , in which the first two values are 1 and 1, and each remaining value is the sum of the previous two values. This is the well-known Fibonacci sequence. For example, for `fibs(7)` the sequence is 1, 1, 2, 3, 5, 8, 13.

When a generator function is called, Python does not execute the function body; instead, it returns an iterator that will execute the function body. When this iterator is invoked for its first value, the iterator starts executing the function body. When execution reaches a `yield`-statement, the iterator delivers the value of the expression in it, as one value in the sequence that the iterator is generating. Then execution of the function body is suspended just after the `yield`-statement.

When the iterator is invoked for another value, execution of the function body is resumed from where it was suspended, with all bindings as they were, as if control had never been transferred away from the function. The function body continues its computation, perhaps to yield more values. The iterator terminates when the function body terminates, and this terminates the sequence of generated values.

Let's compare the `fibs` generator function with similar functions that return the elements of the sequence all at once in a tuple or list. Here's a version that constructs a tuple and returns it.

```
def fibsTuple(n):
    result = ( )
    a = 1
    b = 1
    for i in range(n):
        result += (a,)
        a, b = b, a + b
    return result
```

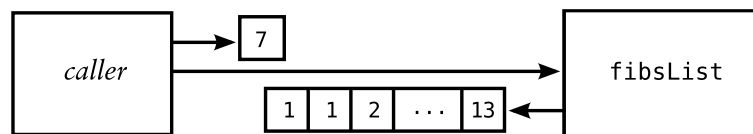
But notice the duplicated effort. Recall that `result += (a,)` means `result = result + (a,)`. Every time around the loop, the function creates a new tuple, a copy of `result` with another value concatenated onto the end. Each tuple but the last is never used again.

Notice the difference in this function, which creates a list rather than a tuple.

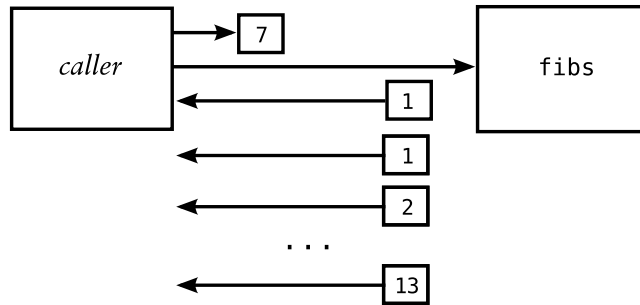
```
def fibsList(n):
    result = [ ]
    a = 1
    b = 1
    for i in range(n):
        result.append(a)
        a, b = b, a + b
    return result
```

This version will probably be more efficient, since it modifies `result` in place rather than creating a whole new data structure each time around the loop. When `n` is large the difference may be significant.

Efficiency aside, both `fibsTuple` and `fibsList` return sequence objects all in one piece, whereas `fibs` returns an iterator that generates a sequence dynamically. We can picture the difference this way: with either `fibsTuple` or `fibsList`, the code that calls the function “pushes” a value of `n` to the function and the function “pushes” a sequence object back.



With `fibs`, the caller “pushes” a value of `n` to the function and then “pulls” values from the function (or, more precisely, from the iterator returned by the function) as it needs them.



Recall the `map` function that we defined in Section 6.4:

```
def map(f, sequence):
    result = ( )
    for a in sequence:
        result += ( f(a), )
    return result
```

This implementation of `map` returns a tuple, in much the same way as `fibTuple` does. It computes a correct result, but it has the same efficiency problem that `fibTuple` has. (Did you notice the efficiency problem when we first presented this definition of `map`?)

We mentioned in Section 6.4 that Python has a version of `map` already defined as a built-in function. That implementation of `map` is actually a generator function, and it works more like this:

```
def map(f, sequence):
    for a in sequence:
        yield f(a)
```

We could include this definition in our own programs in place of the version that returns a tuple: not only is it simpler and more straightforward, but it is likely to be more efficient. There is little point, though; we might as well simply use the built-in `map`.

Like `map`, `filter` and many other Python built-in and library functions that operate on sequences are actually generator functions. They take either static sequences or streams as arguments and produce streams as results.

Just like any other function, a generator function can do more complex computations than those in the simple examples that we have seen so far. Furthermore, a generator function can contain `yield`-statements in more than one place, to combine the results of more than one computation into a single dynamically-generated sequence. Here is an example.

The generator function `combinations`, in Example 7.1 below, takes a non-negative integer n and a sequence of different values, and generates all possible combinations (as tuples) of elements from the sequence taken n at a time. There are three cases:

- If $n = 0$, there is only a single result, the empty tuple.
- If $n > 0$ but the sequence is empty, there are no results. (The Python statement `pass` is merely a place-holder that does nothing.)
- Otherwise, the function recursively generates combinations of two kinds: the combinations that do not include the first element of the sequence, and then the combinations that do.

Example 7.1. Combinations using a generator function

```
def combinations(n, seq):
    if n == 0:
        yield ()
    elif len(seq) == 0:
        pass
    else:
        first = seq[0]
        rest = seq[1:]
        for a in combinations(n, rest):
            yield a
        for a in combinations(n-1, rest):
            yield (first,) + a
```

Take a close look at the code and make sure you see how it works and why it produces the correct result, especially in the recursive cases. Notice that each recursive call works its way closer to a basis case, either because the

parameter `n` becomes closer to zero or because the parameter `seq` becomes shorter.

This computation is a fine example of one that is much easier to do using recursion than with iteration. Try to find a solution using only iteration and see for yourself.

7.3. Endless streams

Using generator functions it is easy to define streams that never end. Here is a simple example: the function `endlessStream` takes any value as an argument, and generates an endless stream of repetitions of that value.

```
def endlessStream(x):
    while True:
        yield x
```

Here is another example. The function `integers` generates successive integers, starting from a given initial value.

```
def integers(n):
    while True:
        yield n
        n += 1
```

Again, the stream generated by the function can continue forever, or at least until `n` becomes too large for Python to store. For all practical purposes, we can consider the stream endless.

We'll use the word “endless” rather than “infinite” to describe streams that never end, to avoid giving the impression that we are somehow computing with objects of infinite size. Doing that isn't possible in computing as we know it, of course. An endless stream of the kind that we can generate in a Python program is simply a sequence that we can take elements from indefinitely, without limit.

It is obvious that `endlessStream` generates an endless stream. In general, though, it may not be obvious from the definition of a generator function whether the stream that it generates is endless or not.

Sometimes termination depends on circumstances external to the program. For example, here's a generator function that yields lines from the program's input, perhaps coming from a person typing at a keyboard. (The built-in function `input` returns the next line from the input, without the terminating “new line” character.) The first empty line terminates the stream generated.

```
def inputLines():
    while True:
        line = input()
        if line == "":
            return # the stream terminates
        else:
            yield line
```

We can't tell by looking at this code whether this generator function will ever terminate. It is entirely under the control of the program's user, not the program.

Even when termination is entirely under the program's control, it is sometimes difficult, or even impossible, to tell whether a stream will terminate. Here's a simple example.

Consider the following simple algorithm, which is sometimes called the Collatz algorithm after the mathematician who proposed it. Start with any positive integer n . Then repeatedly do the following:

- If n is even, divide it by 2.
- Otherwise, multiply it by 3 and add 1.

Stop when and if n reaches 1.

Here is Python generator function that implements the Collatz algorithm, generating the stream of values that n takes on for a given starting value.

```
def Collatz(n):
    yield n
    while n != 1:
        if n % 2 == 0:
            n = n // 2
        else:
            n = 3*n + 1
        yield n
```

For example, with a value of 3 for the argument n , the stream generated is 3, 10, 5, 16, 8, 4, 2, 1.

Believe it or not, as this book is being written in 2014 it is still an unsolved mathematical problem whether the Collatz algorithm terminates for any initial value of n . The sequence has terminated for every n ever tried, but no mathematician has been able to prove that it will terminate for any n .

Termination of computations is an important question in computer science theory, as you will see later in your studies if you are a computer science student. Here, let's just say that it can be difficult or impossible to determine whether an algorithm will always terminate, given a description of the algorithm. This is just one of the many reasons that programming can be challenging.

7.4. Concatenation of streams

Although streams are dynamically generated and may be endless, in some ways they behave much like any other sequences. Let's look in particular at the behavior of streams under concatenation.

Let us define the concatenation of two streams X and Y as another stream, generated in the obvious way: by first generating the elements of X and then generating the elements of Y .

Here is a Python generator function that directly implements that definition:

```
def concat(X,Y):
    for a in X:
        yield a
    for b in Y:
        yield b
```

By the way, notice that `concat` is automatically overloaded for any kind of sequence that a `for`-statement can iterate over: strings, tuples, lists, range objects, and iterators. The function can concatenate two streams, or two static sequences, or one of each. But notice also that each `for`-loop uses an iterator to iterate over `X` or `Y`, whether or not the object is already an iterator. So `concat` is always concatenating two streams in any case. And it always produces a stream, not a static sequence.

What does `concat(A,B)` mean if `A` generates an endless stream? The result is still perfectly well-defined; it just never includes any values from `B`. Then `concat(A,B)` is equal to `A`; that is, the two expressions generate identical streams.

Well, then, is this concatenation operation associative? Certainly it is. It's not hard to see that `concat(A,concat(B,C)) = concat(concat(A,B),C)` for any streams `A`, `B` and `C`, endless or not, as follows.

If all of `A`, `B` and `C` generate sequences that end, then the value of the expression `concat(A,concat(B,C))` is the values generated from `A` followed by (the values generated from `B` followed by the values generated from `C`). Similarly, the value of `concat(concat(A,B),C)` is (the values generated from `A` followed by the values generated from `B`) followed by the values generated from `C`. But here "followed by" is ordinary concatenation of finite sequences, which is associative.

If `A` generates an endless stream, then `concat(A,concat(B,C))` is equal to `A` regardless of `B` and `C`. But then `concat(concat(A,B),C)` is equal to `concat(A,C)` regardless of `B`, and so it is equal to `A` regardless of `C`; thus `concat(A,concat(B,C)) = concat(concat(A,B),C)`. And it is easy to check similarly that the equality holds in the other cases: when `B` generates an endless stream but `A` does not, and (trivially) when only `C` generates an endless stream.

Another way to see that `concat` is associative is as follows. Notice that, when `E` is a call to a generator function, the for-statement

```
for e in E:  
    yield e
```

is just equivalent to the code in the body of the generator function, with arguments to the function substituted into that code appropriately. In effect, the code of the function yields values and the for-statement just passes those values through.

Therefore, the expression `concat(A,concat(B,C))` is equivalent to the concatenation of two code sequences:

```
for a in A:  
    yield a  
  
for b in B:    # all this is  
    yield b    # the meaning  
for c in C:    # of  
    yield c    # concat(B,C)
```

Similarly, the expression `concat(concat(A,B),C)` is equivalent to the concatenation of two code sequences:

```
for a in A:    # all this is  
    yield a    # the meaning  
for b in B:    # of  
    yield b    # concat(A,B)  
  
for c in C:  
    yield c
```

But these two segments of code are clearly equivalent. They must produce the same result, whatever that may be.

So `concat` is associative. Furthermore, it has an identity: the empty stream. Look at the definition again:

```
def concat(X,Y):  
    for a in X:  
        yield a  
    for b in Y:  
        yield b
```

If X generates the empty stream, the first for-statement terminates immediately without yielding anything and the second for-statement generates whatever Y generates, so $\text{concat}(X, Y)$ is equivalent to just Y . Similarly, if Y generates the empty stream, $\text{concat}(X, Y)$ is equivalent to X .

So the empty stream is an identity for `concat`. There is only one empty stream value, although there are many objects that will generate it: the empty string, the empty tuple, any generator function that terminates without yielding anything, and so on. The streams generated by all of these are indistinguishable in Python.

The conclusion: streams are a monoid under `concat`, with the empty stream as identity. By now you should not be surprised at seeing another monoid!

7.5. Programming with streams

Like functional programming, programming with streams gives you a new collection of techniques for you to add to your repertoire. In this section we'll explore a few of them.

Whenever you need to do a computation that generates a sequence of values so that another computation can iterate over that sequence, think of making the first computation generate a stream. Then you can put that computation into a generator function.

For example, generator functions are handy for generating a sequence of values from an object, where the object contains the values we want but is structured differently. Say that we want to do a computation on every character in a file. As we have seen, a file in Python is a sequence of lines, each of which is a sequence of characters. What we want is a different structure: just a sequence of characters.

Here's a generator function that generates that sequence as a stream. The function takes the name of a file as its argument.

```
def chars(file):
    for line in file:
        for char in line:
            yield char
```

Then we might do the main computation something like this:

```
f = open(someFileName)
for char in chars(f):
    main computation using char
```

Here's a slightly more complicated example. Suppose now that the file is a text file, and suppose that we want to produce a sequence of the words in the file. This is a structure that is quite different from the structure of the file itself.

We can use the Python method `split` to separate each line of the file into words. When `split` is applied to a string but called with no other argument, it splits the string at “white space”: that is, sequences of space characters and “new line” characters (and other characters that we don't need to consider in this example, such as tab characters). That's how we would normally divide a line of text into words.

Let's also say that we need to produce the words without any punctuation marks that might be clinging to the beginning or end of them: commas, quotation marks, and so on. Python has another string method that will do this. If `punctuation` is a string containing the punctuation characters that we want to strip away, the value of `word.strip(punctuation)` is `word` with all instances of those characters stripped from the beginning and end. (By the way, `strip` treats the characters in `punctuation` as a set; the description of `strip` in the official Python documentation uses that term.)

Now there's one more complication: if there is white space at the beginning of the line, `split` will produce an empty string in the sequence of “words” that it produces, and we don't want that effect. The same is true for white space at the end of the line, and there will always be at least one “white space” character there, the “new line” character. As it happens, we can use `strip` to

solve this problem too: when called with no arguments other than the string, the method strips “white space” characters.

So here's a generator function that generates a stream of the words in a file. This time we'll put the file-opening operation in the function too, so that the function takes a file name as its argument.

```
def words(fileName):
    punctuation = ".,:;" + ''
    file = open(fileName)
    for line in file:
        words = line.strip().split()
        for word in words:
            yield word.strip(punctuation)
```

As in the previous example, we might do the main computation something like this:

```
for word in words(someFileName):
    main computation using word
```

Let's see how the code of this example might look if we wrote it without the generator function, and combined everything into one big nested loop:

```
punctuation = ".,:;" + ''
f = open(someFileName)
for line in file:
    words = line.strip().split()
    for word in words(f):
        strippedWord = word.strip(punctuation)
        main computation using strippedWord
```

This works just as well, but it doesn't seem as elegant or clear. The main computation, which may itself be rather long, is buried in all the clutter of finding words and stripping punctuation. If all that detail is separated out into a generator function, the part of the program that uses that function can concentrate on the computation that uses the sequence of words.

By definition, streams are generated dynamically, one element after another in time. In fact, the elements of a stream don't necessarily appear at equal intervals in time. Consider the generator function `inputLines` from Section 7.3, for example. Here it is again:

```
def inputLines():
    while True:
        line = input()
        if line == "":
            return # the stream terminates
        else:
            yield line
```

This function generates lines at times that are completely unpredictable, from the program's point of view, assuming that the lines come from a human typing at a keyboard.

Suppose that we want to store input lines typed in that way, but also record with each line the time at which it was entered.

We'll need a way of getting the current time from inside a program. The Python library has a number of functions for getting and operating on times in different formats, but we don't need to concern ourselves with the details of those functions here. Let's just assume that we have a function `time()` that returns the current time, in an appropriate format and with sufficient precision for our purposes.

Then here is a function that generates a stream of two-element tuples, each tuple being the time at which a line was entered and the line itself.

```
def timedInputLines():
    for line in inputLines():
        t = time()
        yield (t, line)
```

This is simple code, but the interesting point about it is the way events unfold in time. Recall that code that uses `timedInputLines` will call it to get an iterator and then "pull" values from that iterator as it needs them. Whenever the code tries to pull a value, `timedInputLines` can't yield a value until it executes the body of its for-statement one more time. That causes the for-statement to try to pull one more value from `inputLines`. And `inputLines` can't yield a line until the user has entered one. All this code will sit waiting, if necessary, for that to happen.

As soon as the user has entered a line (assuming that the line is nonempty), `inputLines` immediately yields it, which causes `timedInputLines` to execute the body of its `for`-statement again. Then `t` gets a time, which is close enough (we hope) to the time at which the user entered the line, and `timedInputLines` yields a tuple back to the code that invoked it.

Computations can use more than one stream at a time. Here's an example: Python has a built-in function called `zip` that takes two static or dynamic sequences `X` and `Y` and generates a stream containing elements from `X` and `Y` paired up, in two-element tuples. For example, the value of `zip((1,2,3),("a","b","c"))` is the stream `(1,"a"), (2,"b"), (3,"c")`.

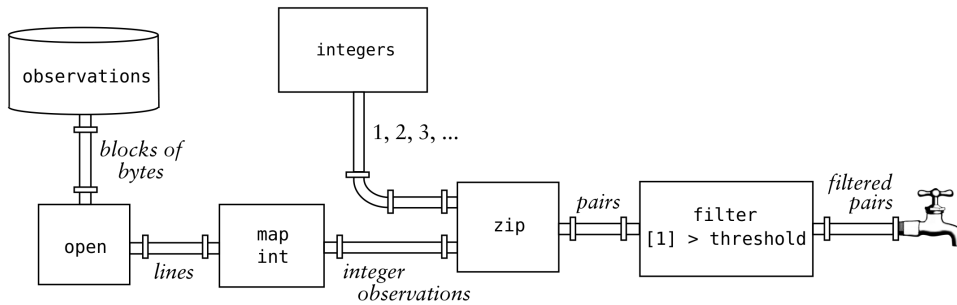
The stream produced by `zip` terminates when either `X` or `Y` terminates. This means that the stream still terminates when either `X` or `Y` is endless but the other is not, and this fact leads to some interesting uses for endless streams. For example, we can number the elements in a sequence by zipping them with the stream produced by the function `integers`. If `y` has the value `("a","b","c","d")`, then the value of `zip(integers(1),y)` is the stream `(1,"a"), (2,"b"), (3,"c"), (4,"d")`.

Here's an application of that technique. Recall the example in Section 1.2 that processed a file of temperature observations. Suppose we need code that finds all the values in the file `observations` that exceeded some given value, say `threshold`. And suppose we don't want just the values: we want each value paired with the position in the file (as a line number) at which it was found. It's not hard to imagine how you would produce this data using ordinary code, with `for`-statements and `if`-statements and so on, but let's do it with streams in a functional-programming style.

As we have seen, `map(int,open("observations"))` converts a stream of lines from the file to a stream of integer observations. We `zip` that stream with a stream from `integers(1)`, to pair each observation with its position in the file, and then filter the resulting stream of pairs. Here's all that code as a single expression:

```
filter(lambda a: a[1] > threshold,
        zip(integers(1),
            map(int,open("observations"))))
```

We can picture the various streams in this expression as pipes through which values flow from one function to the next, something like this:



Now let's see how we would write a `zip` function ourselves if Python didn't already have one. As it happens, the built-in `zip` can zip together any number of sequences, but let's just handle the case of two sequences. So in the definition of `zip(x, y)` we'll want to iterate over `x` and `y` at the same time and generate the first element of `x` paired with the first element of `y`, then the second element of `x` paired with the second element of `y`, and so on.

We immediately run into a difficulty: how do we do the iteration in Python? We'd like to say something like "for each `x` in `x` and for each `y` in `y` at the same time", but Python doesn't have a construct for doing such a thing. If `x` and `y` are static sequences, we can get their corresponding elements by using subscripting — say, `x[i]` and `y[i]` for successive values of `i` — but Python doesn't let us do subscripting on streams.

Python does give us a built-in function, called `next`, that we can call repeatedly to get successive elements of a sequence. The function is called like this:

```
next(iterator, endMarker)
```

Here the argument `iterator` is an iterator that generates successive elements of the sequence that we are interested in. As we have seen, a `for`-statement creates such an iterator automatically if the sequence object is not already an iterator, but to use `next` we must create the iterator explicitly. This is done using the function `iter`, which we can use like a type-conversion function, similar to `int` and `tuple` and so on. If `x` is any static sequence or stream object,

`iter(X)` is an iterator for `X`. It is the same as `X` if `X` is already an iterator, so we can omit using `iter` if we are sure that this is the case; otherwise we must use `iter`, even if `X` is a range object.

The argument `endMarker` is any distinguished value for `next` to return to signal that there are no more elements in the sequence; that is, when `iterator` terminates. Of course, `endMarker` must be different from any value that might be in the sequence. Python programmers often use the value `None`. It's a special value, having a type of its own, that represents "no value"; that is, lack of a value of any other kind.

So here's the pattern of code that we'll use to iterate over a sequence `X` using `next` instead of a for-statement: we'll create an iterator using `iter(X)` and bind it to (say) `it`, then pull values from that iterator using `next(it, None)` until we get the value `None`.

And now let's write a definition for `zip(X, Y)`. We could use `next` to iterate over both `X` and `Y`, but it's a little easier to use a for-statement to iterate over `X` and work an iteration over `Y` into that for-statement. Here is one way to do it. Let's assume that we know that `None` can't be one of the values in `Y`.

```
def zip(X, Y):
    it = iter(Y)
    for x in X:
        y = next(it, None)
        if y == None:
            return
        else:
            yield (x, y)
```

See how `zip` terminates when `X` ends (because the for-statement terminates) or when `Y` ends (because `y` becomes `None` and the `return` statement is executed), whichever comes first. But if both `X` and `Y` are endless, so is the result of zipping them.

Just as we can (in effect) convert a static sequence to a stream by using the `iter` function, we can convert a stream to a static sequence by using `tuple` or `list`. Either of these type-conversion functions, when applied to a stream,

forces the stream to yield all its values at once to be packaged into a static structure. Let us call this process *materializing* the stream.

Why would we ever want to materialize a stream? Perhaps to do some operation that Python allows on static sequences but not on streams. One such operation is subscripting. Another, if we convert the stream to a list, is modifying an element in place. We would also probably want to materialize a stream if we intend to iterate over its values more than once.

Of course, we must beware of trying to materialize a stream that is endless or may be endless. For example, if we try to evaluate `list(integers(0))`, the stream returned by `integers(0)` will not terminate, and so neither will the computation in the `list` function that tries to materialize the stream. Or, if we try to evaluate `tuple(inputLines())`, the `tuple` function will not be able to finish its materializing unless and until the user has typed an empty line. The program will sit and wait until then; it won't be able to process input lines interactively as the user types them.

If we know, or can compute, how many elements of an endless stream we will need, we can always materialize just that many elements. For example, if we need just the first n elements of an endless stream, we can generate a stream of just those elements (see, for example, Exercise 2 at the end of the chapter) and convert the result to a tuple or a list of length n .

In many computations, though, there is no need ever to materialize a stream or a part of it, whether or not the stream is endless. A computation can iterate over the stream directly and control how much of the stream to use, as in our definition of `zip`.

Thus, in many contexts, endless streams are nothing special. In fact, sometimes we can get the simplest code by making our generator functions generate endless streams rather than streams that terminate, letting the code that uses the generator functions determine how much to use.

Here's an example. As we saw earlier in this section, we can number the elements of a sequence by using `zip` on that sequence and a stream generated

by `integers`. Suppose `S` is a stream. Then to number its elements starting from 0, we could write

```
zip(integers(0), S)
```

As we know, `integers(0)` generates an endless stream of integers starting from 0. Now recall that `range(n)` also generates a sequence of integers starting from 0, but only n of them. If we insisted on avoiding endless streams, we could use `range` instead of `integers`, but then we would need to compute the length of `S` in advance. And to do that we would need to materialize `S` first. The resulting code would look something like this:

```
zip(range(len(tuple(S))), S)
```

Not only is this code more complicated and probably less efficient, it won't even work if `S` can't be materialized.

7.6. Distributed processing

As we saw in Section 6.6, parallel processing is making its way into the computer systems that we use every day. But a similar form of processing has been part of our daily lives for years: *distributed processing*, in which computations on separate computers communicate with each other and collaborate to produce a larger computation. We use distributed computing every time we interact with the Internet and the Web, which create a computing environment that acts like a single computer with millions of processors. Applications like Web searching and multiplayer games can easily be single computations distributed over computers all around the globe.

Distributed computing is often described as being based on “message-passing”: for example, computer A sends a message to computer B requesting data, and computer B sends a message back containing the data. But in fact much of the communication in distributed computing is actually based on streams, rather than single messages. For example, computer B may be a server that receives a stream of requests and iterates over that stream, much as in the examples in this chapter, responding to each request. Or the data returned in response to a request, unless it is very small, is likely to be broken into a stream of

“packets”, and computer A will iterate over that stream. Some streams of data are potentially endless, as in audio and video streaming and RSS feeds.

Many computations are distributed simply because data is created or stored in one location but is needed in another. But another reason for distributing a computation is to apply the power of several computers, or even many computers, to a single problem.

Should we always distribute computations if we can? Absolutely not; it depends on the computation, and we should avoid using distributed computing for computations that are not suited to it. Transferring data from one computer to another over a network is a comparatively slow process, and very slow compared to the speed of computation on any one computer. So, even if many computers are available to help with a computation, it makes no sense to transmit large amounts of data to each computer if each computer will do only a small amount of computation on that data.

If each remote computer already has its own data, distributed processing may make more sense. For example, consider a computation on files or web pages that are spread throughout a network. Each computer can do its own searching and aggregating of the data that resides on its own file system, perhaps by using operations like `map` and `filter` and `reduce`, and then send a relatively small quantity of data back to a central computer for aggregating similarly into a final result. Or the aggregating can be done in stages, as with the reduction by \otimes of $a_0 \dots a_7$ that we saw in Section 6.6. Some computers can serve as intermediate aggregators, collecting and aggregating data from several other computers and sending the results on to another computer to be aggregated further.

Again, a bit of mathematical reasoning can help us design this kind of distributed computation for maximum efficiency. For example, if the operation of aggregation is associative, we have more freedom to distribute the operation among machines and do the operation on parts of the data in parallel. We have even more freedom if the operation is commutative; then, for example, an aggregating computer would be able to process results from other computers in whatever order the results may arrive.

Terms introduced in this chapter

stream	yield-statement
generator expression	materialize
generator function	distributed processing

Exercises

1. Experiment with the `fib`, `fibstuple` and `fibslst` functions of Section 7.2. Try to find a value of `n` for which `fibstuple(n)` takes noticeably longer to run than `fib(n)` or vice versa. Repeat with `fibslst` and `fib`. If your computer system gives you a way to measure how long a Python program takes to run, use that; otherwise, just run the Python interpreter interactively and observe.
2. Write a generator function `prefix(n, stream)` that yields the first `n` elements of a stream, or the whole stream if its length is less than `n`. Your function should work whether or not the stream is endless, so materializing the stream first is not a solution.

But notice that the stream that `prefix` returns is not endless in any case, and so it can be materialized if desired.

3. Look again at the `integers` generator function of Section 7.3. Are we really justified in saying that `integers` generates an endless stream? Consider your answers to the first two exercises of Chapter 2.
4. What is the value of each of the following expressions? Be precise in your answers; distinguish between streams and other kinds of sequences, for example. Write code that evaluates the expressions to show that your answers are correct. Assume that `map` is the built-in function, that `double` is as defined in Section 4.3, and that `integers` is as defined in Section 7.3.
 - a. `map(double, range(5))`
 - b. `map(double, integers(0))`

c. `map(integers, integers(0))`

5. In the `timedInputLines` example of Section 7.5, we assumed that all the code of the example would sit waiting for the user to enter each input line. What if the input comes faster than the code can consume it? Perhaps the program is run in such a way that its input comes from another program rather than from a human at a keyboard, for example. How will `timedInputLines` behave?
6. Look again at the averaging-of-temperatures script of Section 1.2. Suppose we have many weather stations at different sites, each taking its own temperature measurements. Could we distribute the operation of averaging to computers at the weather stations, having those computers send their results back to a central computer that would compute one grand average?

Chapter 8

Sets

8.1. Mathematical sets

We introduced the concept of a set back in Section 1.3. In this chapter we'll look at sets in more detail.

Set theory is a central part of advanced mathematics, and set theory can be used to define the foundations of mathematics itself. Set theory, when done with complete formality, is an abstract and difficult subject. Here we'll describe sets much more informally, and concentrate on the concepts of set theory that are most useful in programming. In the current section we'll briefly and informally present definitions that we'll need, along with a few interesting and useful facts about sets.

A set is an unordered collection of different things, which we call the *members* of the set. The things can be whatever we wish to talk about in a given context, such as numbers or names or people or cities. For example, we can define one set as all the people in your family, and another set as all the odd numbers less than 100.

In any particular discussion involving sets, we can explicitly define the *universe of discourse*, meaning all the things, or “elements”, that can be members of the sets that we will be talking about; then the set of all these elements is called the *universal set*. Sometimes, though, we let the universe of discourse be defined implicitly by the things that we happen to talk about, and we don't use the idea of a universal set at all.

In mathematics, the universe of discourse often includes sets themselves, as well as tuples and other composite mathematical objects. If so, sets can contain other sets, as well as tuples and so on.

A set is defined by its members. We consider two sets equal if they have exactly the same members.

We can define a set in several ways. Here are three of the ways:

- By explicitly listing its members. To indicate that a collection is a set, we use curly braces, so that the set of the numbers 1, 3, and 5 is written

$$\{1, 3, 5\}$$

- By giving a property that the members have. An example is “the set of odd positive numbers less than 100”. We could express that phrase as

$$\{ n \mid 0 < n < 100 \text{ and } n \text{ is odd} \}$$

Here the vertical bar “ \mid ” is read as “such that”.

The property selects members from some larger “base” set. That set may be given explicitly as part of the definition:

$$\{ n \mid n \text{ is an integer and } 0 < n < 100 \text{ and } n \text{ is odd} \}$$

Or the base set may be left implicit; it may be defined by context, or we may be meant to assume that it is the universal set.

- By forming the set from other sets using operators, like the union, intersection, and difference operators that we will define below.

There is a set with no members, called the *empty set*, written \emptyset or sometimes $\{\}$.

If an element a is a member of set S , we write $a \in S$. If it is not, we write $a \notin S$. For example, $1 \in \{0, 1, 2\}$ and $1 \notin \{2, 4, 6, 8\}$.

If every member of a set A is also a member of another set B , we say that A is a *subset* of B , written $A \subseteq B$. For example, $\{2, 4\} \subseteq \{1, 2, 3, 4, 5\}$. But notice that, by our definition, it is also true that $\{2, 4\} \subseteq \{2, 4\}$; in fact, any set is a

subset of itself. If we want to say that $A \subseteq B$ but $A \neq B$, we say that A is a *proper subset* of B , written $A \subset B$.

The *union* of two sets A and B is written $A \cup B$, and is another set containing all the elements that are in either A or B or both.

The *intersection* of two sets A and B is written $A \cap B$, and is another set containing all the elements (if any) that are in both A and B .

The *difference* of sets A and B is written $A - B$, and is another set containing all the elements that are in A but are not in B . (It is also sometimes called the “complement” of B with respect to A , and written $A \setminus B$.) For example, $\{1, 2, 3, 4, 5\} - \{2, 4, 6\}$ is $\{1, 3, 5\}$.

Both union and intersection are associative: $A \cup (B \cup C) = (A \cup B) \cup C$ and $A \cap (B \cap C) = (A \cap B) \cap C$. Both operators are commutative as well. These facts are not hard to prove; in fact, they follow directly from the fact that we defined union and intersection using “or” and “and”, and the fact that the latter operators are associative and commutative in logic.

The empty set is the identity for the union operation; therefore, sets are a monoid under union, with the empty set as identity. If there is a universal set, it is the identity for the intersection operation; then sets are a monoid under intersection, with the universal set as identity. If we have no universal set, sets are still a semigroup under intersection.

In mathematics, sets are often infinite in size; that is, they have infinitely many members. For example, mathematicians routinely use the set of all integers and the set of all real numbers. Most of the interesting facts and problems in set theory concern infinite sets.

Furthermore, mathematicians often define sets by giving a property that the members satisfy, but without showing any way to find those members. For example, such a set can be the set of solutions to an equation, like the following set:

$$\{ x \mid x^2 - 5x + 6 = 0 \}$$

Anyone who knows a little algebra knows a method for finding the members of that set, but the definition of the set doesn't tell you the method. Here's another example:

$$\{ (a,b,c,n) \mid a \text{ and } b \text{ and } c \text{ are positive integers, } n \text{ is an integer greater than } 2, \text{ and } a^n + b^n = c^n \}$$

It is not obvious how to find members of this set, if there are any. In fact, until recently no one knew whether this set was empty, nonempty but finite, or infinite; but the mathematician Andrew Wiles proved in a 1995 paper that the equation has no solutions with the given properties, so we now know that the set is empty. (This was the problem of “Fermat's Last Theorem”.)

Compared to mathematicians, we as programmers face obvious limitations on how we can use sets. First, we can't construct infinite sets in our programs. Second, as we noted back in Section 2.3, we must avoid trying to construct sets that are finite but far too large to store in our computers. And, finally, to create a set in a program, we must know a method for constructing it. Even with these limitations, there are plenty of uses for sets in programming, as we will see.

8.2. Sets in Python

Like tuples and lists, sets in Python are data structures that can contain other Python values. The Python term for such a structure is *container*; Python has one more kind of container built into the language, the dictionary, which we will see in Section 9.2.

A Python set, like a static sequence or a stream, is an object that we can iterate over using a for-statement to do something with every member of the set. Any such object is called an *iterable* in Python terminology.

We can construct a set using a *set display*, which has the same form as a list display except with curly braces instead of square brackets on the outside. The form that explicitly lists the set's members is just like the mathematical notation:

```
{ 1, 3, 5 }
```

The other kind of set display is a *set comprehension*, which looks like a list comprehension except with curly braces instead of square brackets on the outside. Here's an example, which is one way to construct the set of odd positive numbers less than 100:

```
{ n for n in range(100) if n % 2 == 1 }
```

A set comprehension is the Python way to construct a set by giving a property that its members have. In Python we can't just state the property, though; we must start by building another set or other iterable — `range(100)` in this case — and then select from it the values with the desired property.

We can also construct a set from a collection of values by using the set type conversion function, writing `set(A)` where A is any iterable. (Now that we have the term “iterable”, we can say that any iterable can also be an argument to the type-conversion functions `tuple` and `list`.) As a special case, the form `set()` constructs an empty set.

Python has equivalents for all the set operators that we defined in the previous section. Here is a table of them:

Table 8.1. Set operators

	math	Python
is a member of	\in	<code>in</code>
is not a member of	\notin	<code>not in</code>
subset	\subseteq	<code><=</code>
proper subset	\subset	<code><</code>
union	\cup	<code> </code>
intersection	\cap	<code>&</code>
difference	$-$	<code>-</code>

Python treats the two-word sequence `not in` as a single operator. The operators `|` and `&` don't look much like the corresponding mathematical operators, but

perhaps you can remember what they mean if you think of them as meaning “or” and “and”; we used those words in our definitions of union and intersection.

Perhaps surprisingly, in Python we can't use `{}` to denote the empty set. This is because `{}` denotes an empty dictionary. It would probably make more sense for `{}` to denote an empty set instead, but Python had dictionaries before it had sets, so the meaning of `{}` is historical. To construct an empty set we must write `set()`.

In Python there is no way to construct a universal set of all possible Python values. Furthermore, in most programming situations, if there is a “universe of discourse” it is either infinite or so large that constructing a universal set from its values would be impossible: imagine the set of all Python strings or the set of all Python integers, for example. But if all the set members in some context in your program are values taken from some rather small finite set, you may be able to use that set as a universal set.

Like lists, Python sets are mutable. Like the `append` method for lists, there is a method `add` that inserts a value into a set. For example:

```
setA = { 2,4 }  
setA.add(3)
```

After those statements are executed, `setA` contains the values 2, 3, and 4. If 3 had already been in `setA`, the second statement would have had no effect.

Python has other methods and functions that operate on sets; we'll mention just two more here. For a set `A` and a value `x`, the statement `A.discard(x)` removes `x` from `A`; if `x` is not in `A`, the statement has no effect. And the function `len` can be applied to sets as well as sequences: `len(A)` is the number of elements in `A`.

As in mathematics, sets in Python are unordered. This means that if you iterate over a set, you will get all the members, but in no particular order. For example, if you execute these statements:

```
cities = { "London", "Paris", "Vienna", "Istanbul" }  
for place in cities:  
    print(place)
```

you might see this:

```
Paris  
London  
Istanbul  
Vienna
```

or you might see the names of the cities in a different order.

Python sets are designed to make testing for membership particularly efficient. This is fortunate, because Python must use membership testing not only to do the `in` and `not in` operations, but also to do intersections and unions. To evaluate `A & B`, Python does something like this: it tests each member of `A` to see whether it is also in `B`. Similarly, to evaluate `A | B`, Python does something like this: it starts with the members of `A` and then tests every member of `B`, adding the member to the result only if it is not in `A`.

Python uses a technique called *hashing* to implement sets. The details of hashing are beyond the scope of this book, but let's just say that hashing involves using the value of an object to compute the object's location in a data structure. Therefore, to see whether a value is a member of a set, Python simply does the hashing computation to see whether there is an object with that value at the corresponding location. If the set is large, this computation is much faster than comparing the given value with each member of the set.

But, once an object is in a set, the object's value can't be allowed to change, since then the value and the object's location in the set wouldn't correspond. Therefore, a Python set can contain only immutable objects. In particular, a set can't contain a list.

Similarly (and unfortunately), a Python set can't contain another Python set. Fortunately, Python has another kind of data structure called a “frozen set”: it behaves like a Python set except that it is immutable. We have seen that tuples behave like lists except that tuples are immutable; the relationship between sets and frozen sets is the same.

To construct a frozen set, we use the type conversion function `frozenset`; its argument can be a set, or any other collection of values such as a sequence. To construct an empty frozen set, we write `frozenset()`.

So by converting a set into a frozen set, we get a structure that we can include in another set. For example, in mathematics the combinations of the numbers $\{0,1,2\}$ taken two at a time is a set of sets:

$\{\{0,1\}, \{0,2\}, \{1,2\}\}$

In Python, we can't write it quite that way: the inner sets need to be frozen. We need to write something like this:

```
{ frozenset(c) for c in ((0,1),(0,2),(1,2)) }
```

8.3. A case study: finding students for jobs

Now let's pause for a short case study, to apply what we have learned so far to a simple data-processing problem.

Suppose that a company is recruiting students at a university to give them jobs after they graduate, and suppose that the company wants to interview fourth-year students who are either computer science or electrical engineering students and who have a grade average of B or better.

Let's also suppose that the university has data files that will help us find such students. The file `cs` contains the names of all the computer science students, and the file `ee` contains the names of all the electrical engineering students. The file `year4` contains the names of all the students who are in their fourth year of study, and the file `goodGrades` contains the names of all the students with a grade average of B or better.

Let's see how we can find the names of all the appropriate students. This is a very easy programming problem, but let's try to find the very best solution that we can. We'll start by showing how a beginning programmer might approach the problem, and then show how the solution changes as we apply progressively more of the ideas that we have seen so far in this book.

Very early in their programming careers, beginners learn a stereotyped structure for a program:

1. Read all the inputs.
2. Do the computation.
3. Display the results.

For some programming problems this approach is far too simplistic, but it will work perfectly well for the current problem.

So let's first consider the task of reading all the inputs. A true beginner might use another stereotyped strategy: to read all of a file, read lines one at a time until there aren't any more, using a while-statement with calls to `readLine` and a test for the empty string, as we saw in an example in Section 3.3.

Insight #1: we don't need to do all that. Reading the file using a for-statement is much easier, as any reader of this book surely knows by now.

So now one way to proceed is to read lines from a file and collect the names in a data structure. The first structure to come to mind may be a list, especially if the programmer is used to programming in a language that isn't quite as high-level as Python is. So our strategy might be to start with an empty list, and add each name from the file as we read the lines, like this:

```
names = [ ]
for line in file:
    names.append(line.strip())
```

Insight #2: we don't need to do all that. This kind of computation is exactly what a list comprehension is meant for. We can write this instead:

```
names = [ line.strip() for line in file ]
```

We can use such a statement to read each of our data files:

```
year4 = [ line.strip() for line in open("year4") ]
cs = [ line.strip() for line in open("cs") ]
ee = [ line.strip() for line in open("ee") ]
goodGrades = [ line.strip() for line in open("goodGrades") ]
```

Insight #3: there's a lot of common computation here, which we can separate out into a function. Doing this simplifies the code and is generally good practice anyway. Writing the function is particularly easy in Python, because we can still use a comprehension and simply let the function return the resulting list.

```
def listOfNames(filename):
    return [ line.strip() for line in open(filename) ]

year4 = listOfNames("year4")
cs = listOfNames("cs")
ee = listOfNames("ee")
goodGrades = listOfNames("goodGrades")
```

Now that we have all the data in data structures in the program, we can proceed with the main computation. The statement of our problem asks us to find “fourth-year students who are either computer science or electrical engineering students and who have a grade average of B or better”. Here is one obvious way to translate that requirement into Python:

```
candidates = [ ]
for student in year4:
    if (student in cs or student in ee) \
        and student in goodGrades:
        candidates.append(student)
```

Insight #4: we can do the same computation more efficiently using sets. The computation does most of its work with the `in` operator, which Python can do more efficiently on sets than on lists. We'll need to change our code that reads input so that it creates sets instead of lists, but the changes are obvious and we'll omit them for the moment.

```
candidates = set()
for student in year4:
    if (student in cs or student in ee) \
        and student in goodGrades:
        candidates.add(student)
```

Insight #5: we don't need to do all that. Again, we don't need to initialize a data structure and then add values to it one at a time: we can use a comprehension.

```
candidates = { student for student in year4
                if (student in cs or student in ee)
                    and student in goodGrades }
```

Insight #6: we don't even need to do all that. Now that we have all our names in sets, why not just use set operations instead?

```
candidates = year4 & (cs | ee) & goodGrades
```

Our task is to find “fourth-year students who are either computer science or electrical engineering students and who have a grade average of B or better”. How could we express the computation any more simply and clearly? And it's efficient, too.

So, once we put the pieces together and add an obvious for-statement to display the results of the computation, here is our program. It's short and simple:

Example 8.1. Finding job candidates using set operations

```
def setOfNames(fileName):
    return { line.strip()
            for line in open(fileName) }

year4 = setOfNames("year4")
cs = setOfNames("cs")
ee = setOfNames("ee")
goodGrades = setOfNames("goodGrades")

candidates = year4 & (cs | ee) & goodGrades

for student in candidates:
    print(student)
```

For comparison, you might like to write out the program that we would have now if we had accepted the beginner's solution at each step of the development.

8.4. Flat files, sets, and tuples

Let us return to the theme of flat files that we introduced in Section 5.3. As we have seen, many flat files represent sets or sequences of values. Often these values are simply numbers or strings, as in the case study of the previous section or in Example 1.1. In each of those programs, the data files contained names, one name per line, but conceptually each file was a set of names.

More often than not, a program that takes data from such a file will use all the data in the file. In developing such a program, an obvious first step can be to write code to get all the data into a data structure in the program. The function `setOfNames` in the previous section did just that: it took as a parameter the name of a file, and returned a set of the names in the file. Here is that function with a couple of minor changes.

```
def setOfValues(fileName):  
    file = open(fileName)  
    return { line.strip() for line in file }
```

Here the function has a slightly more generic name, to be suitable for use when the values in the file are strings of any kind, not necessarily names. (We might have been tempted to use just the name “set”, but if we did we wouldn't be able to use the built-in set-constructing function with that name.) We put the call to `open` in a separate statement, but we could just as well have included it in the set constructor, as we did in `setOfNames`. Recall that in this code, as in all examples in this book, we omit the code that would handle situations in which the file cannot be opened.

The function `setOfValues` constructs a set from the values in the file, but we can just as easily construct a list:

```
def listOfValues(fileName):  
    file = open(fileName)  
    return [ line.strip() for line in file ]
```

or a stream:

```
def streamOfValues(fileName):  
    file = open(fileName)  
    return ( line.strip() for line in file )
```

or a tuple, constructing a stream and then materializing it:

```
def tupleOfValues(fileName):
    file = open(fileName)
    return tuple(( line.strip() for line in file ))
```

We'll discuss multisets in Section 9.5, and there we'll see what you can do if you want to treat the data in a file as a multiset, rather than as a set or a sequence.

Now, what if a data file contains lines that are divided into fields? Let's consider the common case of CSV files, the “comma-separated values” files that we saw in Section 5.3.

As with files of single values, we'll often want to read the whole file and put all of its data into a data structure. For a CSV file, the obvious kind of data structure is a set, or possibly some kind of sequence, of tuples.

We can split a line into its fields by using the `split` method:

```
line.split(",")
```

So it might appear that we could create the set of tuples that we want using a comprehension like this:

```
{ line.strip().split(",") for line in file }
```

Unfortunately, that solution isn't quite right. It happens that the Python method `split` produces a list, not a tuple. And we can't construct a set of lists, because lists are mutable.

The solution is to convert each list that we get from `split` to a tuple, and construct a set from those tuples:

```
{ tuple(line.strip().split(",")) for line in file }
```

And here is the obvious function to read a file and give us the set of tuples that it represents:

```
def setOfTuples(fileName):
    file = open(fileName)
    return { tuple(line.strip().split(","))
            for line in file }
```

Of course, we can just as easily write a function to construct a list, or stream, or tuple of the tuples that we get from the data in the file. We'll assume that we have the functions `listOfTuples`, `streamOfTuples`, and `tupleOfTuples` if we need them; the definitions are obvious.

Once we have the tuples in any of these data structures, we can iterate over all the tuples using a for-statement. For example, suppose the file `directory` is a CSV file in which each line contains a name, a telephone number, and an email address. We can use code like the following to get each tuple, unpack the tuple, and then do something with the fields:

```
directory = setOfTuples("directory")
for entry in directory:
    (name, phone, email) = entry
    ...
```

We can also incorporate the unpacking into the for-statement's header. A for-statement binds a name to a value from the stream that it is iterating over, much as an assignment statement binds the name on its left-hand side to a value. Python lets us bind the elements of a tuple to a comma-separated sequence of names, in a for-statement header just as in an assignment statement:

```
directory = setOfTuples("directory")
for (name, phone, email) in directory:
    ...
```

We can also use the `for (...) in ...` form in a comprehension. For example, suppose we have the same file, but we are only interested in the names and email addresses. Here is one way to construct a set of the corresponding two-element tuples from the file:

```
directory = setOfTuples("directory")
emailDirectory = { (entry[0], entry[2])
                  for entry in directory }
```

But the “unpacking” construct lets us give names to the tuple elements, which makes the intent of the code clearer:

```
directory = setOfTuples("directory")
emailDirectory = { (name, email)
                   for (name, phone, email)
                   in directory }
```

Using a comprehension with filtering (the “for ... in ... if ...” form) we can easily select only certain tuples, or certain fields from certain tuples, as we read the file. For example, consider the example of the previous section again. Suppose that instead of being given the files `cs` and `ee`, we are given a single file `students` with the names of all students paired with their major fields of study, something like this:

```
John Baez,math
Jude Collins,CS
Joan Denver,CS
Joan Denver,EE
Roberta Dylan,physics
...
```

A student declaring more than one major field of study would have more than one line in the file, like Joan Denver above. Thus the file represents a relation but not a mapping, but we don't care about the distinction in this case.

We can construct the sets `cs` and `ee` easily as follows:

```
students = setOfTuples("students")
cs = { name for (name,major) in students
      if major == "CS" }
ee = { name for (name,major) in students
      if major == "EE" }
```

Notice how each comprehension reads easily and has an obvious meaning, but is also concise and easy to write for the amount of work it does.

We can simplify the code a bit by constructing a single set `csOrEE` containing all the computer science and electrical engineering students, instead of the two sets `cs` and `ee`:

```
students = setOfTuples("students")
csOrEE = { name for (name,major) in students
          if major == "CS" or major == "EE" }
```

And then we don't need the set-union operation in the main computation. We can write this instead:

```
candidates = year4 & csOrEE & goodGrades
```

Having made those changes, here's one more easy improvement we can make. Since we aren't going to use the structure `students` more than once, we don't need to materialize it as a set at all: we can leave the collection of tuples as a stream, and we can let the comprehension simply iterate over that stream.

With all these changes, our program might appear as shown in Example 8.2.

Example 8.2. Job candidates again, with different input files

```
def setOfNames(fileName):
    return { line.strip()
            for line in open(fileName) }

def streamOfTuples(fileName):
    return ( line.strip().split(",")
            for line in open(fileName) )

year4 = setOfNames("year4")
csOrEE = { name for (name,major)
          in streamOfTuples("students")
          if major == "CS" or major == "EE" }
goodGrades = setOfNames("goodGrades")

candidates = year4 & csOrEE & goodGrades

for student in candidates:
    print(student)
```

You can probably see how to adapt the program to other forms that the input files might have (see Exercise 3, for example).

8.5. Other representations of sets

Even though Python has sets and a convenient notation for using them built into the language, we don't need to use a Python set as the representation for every collection of data that we view as a set conceptually. We have just seen an example: the names in the file `students` in the previous section. Conceptually those names are a set, because we don't care about their order and we don't expect duplicates. But their representation in the file is as a static sequence, and as seen by our program they are a stream. Both representations are perfectly appropriate.

Our representation of a set should depend on the operations that we need to do with the set or its elements. A representation as a sequence is perfectly good if all we need to do is to iterate over all elements of the set, as with the file `students` and the files of Examples 1.1 and 1.2. We had no need to convert the data in those files to Python sets.

Python sets are better representations of conceptual sets if the operations will be testing values for membership or computing unions and intersections. Not only does Python have convenient and elegant syntax for these operations, but the operations are implemented quite efficiently by the Python interpreter.

In a programming language without sets and set operations built in, programmers may need to implement those features themselves. Computer scientists have invented a variety of data structures for sets and algorithms for set operations. These are beyond the scope of this book, but if you are a student of computer science you will probably see them later in your studies. Fortunately, many programming languages come with ready-made libraries that implement sets and set operations. The libraries are not as convenient as the built-in Python features, but they do the job.

But, in Python or in any other language, it may not be appropriate to use such heavy-duty machinery just because it is available. Here's another example of an alternate set representation. Suppose that for some particular purpose in a program we need a set of n different objects or values. The only property that they need to have is that they are all different. The only operation on

them will be to test whether one value is the same as another. We won't construct subsets of them, just use them individually.

It may help us clarify our thinking about the problem to think of the objects or values as a set while designing the program. But in the implementation, the best representation of the set members may be simply the integers from 0 to $n-1$. The set as a whole might be represented as a Python range object, if we need to iterate over all the members; if not, there may be no need to represent the set explicitly at all. What could be simpler?

Terms introduced in this chapter

member	intersection
universe of discourse	difference
universal set	container
empty set	iterable
subset	set display
proper subset	set comprehension
union	hashing

Exercises

1. Write a Python function that takes as its argument a nonempty set of file names; the corresponding files contain names of people, as in the examples of Section 8.2. Your function will return the set of the names that are in all of the files. For example, the files may be a teacher's lists of the students in class on different days, and your function will compute the names of the students with perfect attendance. Make your code as simple and elegant as you can.
2. Write a set comprehension that uses an integer value n to produce the set of different factors of n ; that is, the set of integers that divide n evenly, not including 1 and n itself. Then use this comprehension to define a function `isPrime(n)` that returns `True` if n is prime and `False` otherwise. An integer is prime if it has no factors.

Can you rewrite your `isPrime` so that it tests only the prime factors of n ? It is easy to see that if n has any factors at all it must have prime factors. Also, it is also not hard to see that if n has any factors, at least one must be less than or equal to the square root of n ; take advantage of that fact too to cut down the amount of work that your function needs to do.

Which version of `isPrime` is more efficient? Experiment by timing both versions.

3. Write a version of the program of Examples 8.1 and 8.2, assuming that the inputs are now these CSV files:
 - `students`, in which each line has three fields: a name, a major field of study, and number of years of study (an integer, 4 for a fourth-year student).
 - `grades`, in which each lines has two fields: a name and a grade average (a number that may contain a decimal point, on a scale of 0.0 to 4.0, so that 3.0 would represent a B average).

If you find that this exercise is almost too simple to bother with, once you have seen Examples 8.1 and 8.2, perhaps that's the point of the exercise.

4. For what purposes in a program might you need a set of n objects or values that are all different but need have no other properties, as described at the end of Section 8.5? Try to think of a variety of examples.

Chapter 9

Mappings

9.1. Mathematical mappings

We introduced the concept of a mapping in Section 1.3. In this chapter we'll look at mappings in more detail. As we did with sets, we'll describe mappings very informally, concentrating on the properties of mappings that will be most useful to us as programmers.

A mapping is a set of ordered pairs in which no two ordered pairs have the same first element. Therefore, a mapping associates a specific second element with each value that is the first element of some ordered pair in the set. In mathematics, “mapping” is a synonym for “function”, and we can use the notation of functions to define a mapping. For example, to say “ $f(3) = 7$ ” is to say that the ordered pair $(3,7)$ is in the mapping f .

As with sets, there are a number of ways to define a mapping. One way is to list the ordered pairs explicitly, like this:

$$\{ (1, 1), (2, 2), (3, 6), (4, 24) \}$$

A common variation on this notation uses the operator \mapsto to denote each ordered pair, to emphasize that the first element is mapped to the second:

$$\{ 1 \mapsto 1, 2 \mapsto 2, 3 \mapsto 6, 4 \mapsto 24 \}$$

In this notation, each pair $a \mapsto b$ is called a *maplet*.

Mathematicians use a variety of other formal and informal notations to define mappings. For example, here are three ways to define f to be a factorial function:

$$f(n) = n! \text{ for all positive integers } n$$

$$f = \lambda n . n! \text{ for all positive integers } n$$

$$f = \{ n \mapsto n! \text{ for all positive integers } n \}$$

For a given mapping f , its *domain*, written $\mathbf{dom} f$, is the set of values that are first elements of ordered pairs in the mapping. Its *range*, written $\mathbf{ran} f$, is the set of values that are second elements of ordered pairs in the mapping.¹ Notice that, as in the case of the factorial function, the domain and range of a mapping can be infinite sets.

Sometimes the domain of a mapping is obvious from context. Sometimes, though, we must specify what the domain is to avoid ambiguity. Take, for example, the expression “ $\lambda x . x + 1$ ”. Does it define the successor function for non-negative integers, or the function that adds 1 to any real number, or something else? Depending on our choice of domain, we get different functions, because the sets of ordered pairs are different.

In mathematics as in programming, a function can take more than one argument. There are two common ways in mathematics to construct such a function. The first is to define the function as a mapping from a tuple of values to a single value. Consider, for example, a function that maps two numbers to their product:

$$p(x,y) = xy$$

Then p could be defined as follows, using the maplet notation:

$$p = \{ (x,y) \mapsto xy \}$$

With this definition, $p(x,y)$ can be interpreted as shorthand for $p((x,y))$, so that, for example, $p(2,3) = 6$ as one would expect.

¹In mathematics there are a variety of terms and definitions for these and related concepts, but these are the definitions that we will use in this book. Be aware, though, that you might see somewhat different vocabulary in other books.

The second way to define a function of more than one argument is as a chain of mappings, each of which maps an argument to a function of the remaining arguments. Here is how p would be defined in this way:

$$p = \{ x \mapsto \{ y \mapsto xy \} \}$$

A function that is defined as a chain of mappings in this way is said to be *curried*, after the mathematician Haskell Curry.²

With this definition, the value of $p(x)$ is the function of one argument that multiplies its argument by x . Then $p(x,y)$ is interpreted as shorthand for $(p(x))(y)$, so that again $p(2,3) = 6$ as expected.

But notice that, if p is defined as a curried function, $p(x)$ also has a useful meaning by itself: it is like the Python `multiplyBy` function that we saw in Section 4.3 when we discussed partial application. In general, whenever a function is defined using currying, partial application of the function to its arguments (left to right) is defined automatically.

Even though we define a mapping as a set, the usual set operators aren't often very useful. For example, consider $f \cup g$ where f and g are mappings. The expression denotes a set of ordered pairs, but not a mapping except in the special case that the domains of f and g don't overlap. The expressions $f \cap g$ and $f - g$ do denote mappings, but not very useful mappings in most situations.

There are mathematical operators that apply specifically to mappings, and we'll describe three of them here. One is the *domain restriction* operator. For a mapping f and a set A , $f|_A$ is f with its domain restricted to A . We can define it this way:

$$f|_A = \{ a \mapsto b \mid a \mapsto b \in f \text{ and } a \in A \}$$

Another mapping operator is called *overriding union* or just *override*. For two mappings f and g , $f \oplus g$ is f overridden by g ; it is a mapping that agrees with

²Yes, feel free to make the obvious puns. Chutney, anyone?

g everywhere on the domain of g and agrees with f otherwise.³ We can define \oplus like this:

$$f \oplus g = g \cup (f \upharpoonright_{\text{dom } f - \text{dom } g})$$

Mapping overriding isn't given much attention in mathematics, but it is central to computation. We can describe a computer's main memory as a mapping from locations to values stored at those locations, and any operation that stores values in one or more memory locations is an overriding of that mapping. Similarly, elements of a data structure are often accessed by subscripts or keys. Then the data structure forms a mapping, and any operation that stores values into the data structure overrides the mapping. We will see examples from Python in the next section.

Although the fact may not be obvious at first glance, the override operator is associative: $(f \oplus g) \oplus h = f \oplus (g \oplus h)$ for all mappings f , g , and h . And there is an empty mapping, the empty set, and it acts as an identity for \oplus . Therefore (surprise!) mappings are a monoid under \oplus , with the empty mapping as the identity. But \oplus is certainly not commutative, because it is asymmetric: maplets in the second operand take precedence over maplets in the first operand.

We can sometimes use the associative property of overriding to improve the efficiency of updates to large mappings, as found (for example) in databases, or the efficiency of updates to mappings held at remote locations in distributed systems. A sequence of updates g_0, g_1, \dots, g_n to a mapping f replaces f by $((f \oplus g_0) \oplus g_1) \oplus \dots \oplus g_n$. Especially if all the updates are relatively small compared to f , it can sometimes be more efficient to precompute the whole “batch” of updates $g_0 \oplus g_1 \oplus \dots \oplus g_n$ and then override f with the result. Notice that the previous expression contains no parentheses; again by associativity, we can group the updates in any way we like, or construct a “big \oplus ” operation that combines them all. Since overriding is not commutative, though, we must take care to keep the updates in the proper order.

³ The symbol \oplus is from the Z notation. Z is a mathematical notation, developed at Oxford University, for writing specifications of programs.

A third operator on mappings is the *composition* operator. For two mappings f and g , their composition is written as $g \circ f$ and is defined as follows:

$$(g \circ f)(x) = g(f(x)).$$

In other words:

$$g \circ f = \{ (a, c) \mid (a, b) \in f \text{ and } (b, c) \in g \}$$

One can read $g \circ f$ as “ g after f ”.

Notice that for a value x to be in the domain of $g \circ f$, not only must x be in **dom** f , but $f(x)$ must be in **dom** g .

A mapping f can be composed with itself, which is most useful when **ran** $f \subseteq$ **dom** f ; in fact, we often compose a mapping with itself more than once. Instead of $f \circ f$ we often write f^2 , for $f \circ f \circ f$ we often write f^3 , and so on for higher “powers” of f .

9.2. Python dictionaries

We have already seen Python constructs that act like mathematical mappings. Most obviously, a Python function that returns a value is a mapping. But static sequences — tuples, lists, and even strings — act like mappings too. Recall that an element of a static sequence s can be accessed by position: $s[i]$ where i is an integer value in the range 0 to $\text{len}(s)-1$ inclusive. Therefore, we can use s as a mapping having that range as its domain.

A Python dictionary is another kind of static container that acts like a mapping, but whose domain is not necessarily a range of integers; the first element of each ordered pair can be of any of a wide variety of Python types.

To construct a dictionary, we can use a *dictionary display* that shows the ordered pairs explicitly. A dictionary display looks like a set display containing ordered pairs, which are written as two values separated by a colon. Here is an example that maps several English names of numbers to their integer values:

```
numberValue = { "one":1, "two":2, "three":3 }
```

In this context the colon acts like the maplet operator \mapsto .

The first element of the ordered pair is called a *key*. To get the value associated with a given key in the dictionary, we use an expression of the same form as an expression that selects an element of a sequence by position: the name of the dictionary, followed by a key in square brackets. Thus, in the example above, `numberValue["two"]` would have the value 2.

Dictionary keys can be of almost any Python type; for example, Python programmers frequently use strings as dictionary keys, as in the example above. But Python uses hashing (Section 8.2) on the key of each ordered pair to store the pair in the dictionary, so a dictionary key, like a set member, must be an immutable object. The value associated with the key, on the other hand, can be any Python value.

Python has dictionary comprehensions, which look like set comprehensions except containing ordered pairs using colons. This comprehension, for example, associates each integer from 0 to 49 with its factorial (Section 4.2).

```
{ n:factorial(n) for n in range(50) }
```

Python has a number of operations on dictionaries; we'll describe only a few of them here. For any dictionary `d`, `items(d)` is an iterator that yields the key-value pairs in `d` as two-element tuples, `keys(d)` is an iterator that yields the keys of `d` (that is, the domain of `d`), and `values(d)` is an iterator that yields the values associated with the keys of `d` (its range). To get the number of ordered pairs in `d`, we write `len(d)`.

The assignment statement `d[k] = v` associates the value `v` with the key `k` in `d`; it overrides any existing value associated with `k` if there is one. In mathematical notation, it replaces `d` with `d ⊕ { k ↦ v }`. If `e` is another dictionary, `d.update(e)` overrides `d` with all the ordered pairs in `e`; it replaces `d` with `d ⊕ e`.

For the programmer's convenience, a few dictionary operations use the keys in a dictionary where one might expect the ordered pairs to be used instead. The Boolean expression `k in d` tests whether `k` is a key for some ordered pair in `d`. As with sets, the `in` operator uses hashing for efficiency. The expression

`k not in d` works similarly. And an operation that iterates over a dictionary, such as a for-statement, creates an iterator over the dictionary's keys rather than over its key-value pairs. (As in iterating over a set, the order is unpredictable). In the following example, if `codes` is a dictionary whose values are strings, `codesString` becomes the concatenation of those strings in some order.

```
codesString = ""
for k in codes:
    codesString += codes[k]
```

Programmers frequently construct dictionaries from data in flat files, using code similar to the code that we presented in Section 8.4 for constructing sets from flat files. In fact, we can conveniently use the generator function `streamOfTuples` from Example 8.2 to help in constructing dictionaries from CSV files. Here is the function again:

```
def streamOfTuples(fileName):
    return ( line.strip().split(",")
            for line in open(fileName) )
```

Suppose that `addressBook` is a CSV file whose lines contains names of people and their email addresses. Here is one way to construct a dictionary that maps names to email addresses:

```
address = { name : email
            for (name,email)
            in streamOfTuples("addressBook") }
```

And here is one way to construct mappings in both directions, while reading the file only once for efficiency:

```
address = {}
owner = {}
for (name,email) in streamOfTuples("addressBook"):
    address[name] = email
    owner[email] = name
```

Now suppose that we want to construct a dictionary in which we use pairs of values to select elements. For example, suppose that `populations` is a CSV file in which each line is a triple: the name of a country, the name of a city in

that country, and the city's population. Sometimes we need both the city name and the country name to find the correct population figure; for example, Perth in Australia is not the same place as Perth in Scotland. Let's see how to construct from the file a dictionary that maps a country and a city to the city's population.

Just as in constructing a mathematical mapping with two arguments, there are two obvious ways to construct the dictionary that we want. The first way is to use a 2-tuple of a country and a city as a key. Here's how we might do it:

```
population = { (country,city) : pop
               for (country, city, pop)
               in streamOfTuples("populations") }
```

Then, for example, we could get the population of Perth in Australia with the expression `population["Australia","Perth"]`, assuming that the file `populations` contained a line with the corresponding data. Recall that the comma between the two strings creates a tuple from them; we don't need parentheses.

If you have written programs in other programming languages, you may have encountered a data structure called a “two-dimensional array”. It may appear that `population` is such a structure, accessed by a country in one “dimension” and by a city in the other. But it isn't: it is a “one-dimensional” structure accessed by a single value, a 2-tuple.

In Python, the more usual way to construct a dictionary accessed by pairs of values is like defining a curried function: by defining a dictionary of dictionaries. In the current example, we might define a dictionary whose keys are country names, and whose associated values are dictionaries that map city names to populations. Here is one way to do it. Notice that we must iterate over the tuples twice: the first time: the first for-statement finds the set of the countries in the data and constructs all the first-level dictionaries.

```
population = {}
tuples = setOfTuples("populations")
for country in { co for (co, city, pop) in tuples }:
    population[country] = {}
for (country, city, pop) in tuples:
    population[country][city] = pop
```

Then we would get the population of Perth in Australia with the expression `population["Australia"]["Perth"]`.

You can probably see what you would need to do to construct a dictionary accessed by more than two values.

9.3. A case study: finding given words in a file of text

Let's pause for another case study, to put together a number of the concepts that have been presented so far. We'll present a programming problem and then see how we might choose discrete-mathematical structures for various parts of the solution.

Suppose we have been given the job of writing the following program:

I need a program that will search an electronic document for a number of words. For each of those words, the program will display the word and the line number of every line in the document on which the word occurs.

For current purposes, the words will be in a file named “targets”, one word per line, and the document will be in a file named “document”.

Perhaps the person making the request is an author creating an index for a book, or a researcher looking for particular terms in a web page. We'll call this person “the client”. Let's analyze in discrete-mathematical terms the specifications that the client gave us.

The words that we will search for are apparently a set.

The information that the client wants is a mapping from words to collections of line numbers. What are the collections? The client says “the line number of every line in the document on which the word occurs”. If a word occurs twice on a line, the client apparently doesn't want to see the line number twice. So let's make each collection a set so that it won't contain duplicates.

One way to look at the document is as a sequence of lines, each of which is a sequence of words separated by white space. Another way is simply as a sequence of words separated by white space, as we did in Section 7.5. But that isn't enough information in this case, because we'll need to know the number of the line on which each word occurred. So let's consider a third possibility: we'll treat the document as a sequence of words, each with an associated line number; in other words, a sequence of ordered pairs. In fact, we could treat the document as a set of such pairs — we would get the same result — but, since we anticipate that the document may be very long, we'll plan to treat it as a stream of ordered pairs, generated dynamically as we read the document.

We have already seen how to do some parts of our computation. To construct the set of “targets”, the words that we are looking for, we can use a simple set comprehension like one that we used in Section 8.3:

```
targets = { line.strip() for line in open("targets") }
```

In Section 7.5 we defined a generator function to generate a stream of words from a file, removing leading and trailing punctuation from each word:

```
def words(fileName):
    punctuation = ".,:;" + ''
    file = open(fileName)
    for line in file:
        words = line.strip().split()
        for word in words:
            yield word.strip(punctuation)
```

We can easily modify this function so that it counts lines as it reads them, and so that every value that it yields is a two-element tuple of a word and a line number:

```
def wordsAndLineNumbers(fileName):
    punctuation = ".,:;" + "'"
    lineno = 0
    file = open(fileName)
    for line in file:
        lineno += 1
        words = line.strip().split()
        for word in words:
            yield (word.strip(punctuation), lineno)
```

A Python dictionary is ideal for representing the mapping that we are constructing; words in `targets` will be the keys of that dictionary. We call it `linesFoundOn`, and initialize it to map each word in `targets` to an empty set, using a dictionary comprehension:

```
linesFoundOn = { t : set() for t in targets }
```

Then the main computation is very easy using set operations:

```
for (word, lineno) in wordsAndLineNumbers("document"):
    if word in targets:
        linesFoundOn[word].add(lineno)
```

Now we display the results. The only hard part is to format each set of line numbers in an appropriate way. Let's assume that we have a function named `formatted` to do that; we'll define `formatted` later. Using that function, it's easy to display the results that the client asked for:

```
for t in targets:
    print(t + " " + formatted(linesFoundOn[t]))
```

The only job left is to define `formatted`, and first we need to decide how we want to format each set of numbers. The client didn't say anything about displaying line numbers in any particular order, but we can guess that ascending order might be most convenient for the client. Python has a built-in function named `sorted` that takes any iterable and returns its values sorted, as a list; we'll use that function. So the definition of `formatted` will use the expression `sorted(numberSet)`, where `numberSet` is the parameter of `formatted`.

Python also has a built-in function named `join` that takes any sequence of strings and concatenates them into one string, separated by another given string. The syntax is `separator.join(strings)`. Let's use a comma followed by

a space as a separator. So far we have `sorted(numberSet)`, which is a list of numbers; we can convert it to a sequence of strings by writing this:

```
map(str, sorted(numberSet))
```

Then we can define `formatted` like this:

```
def formatted(numberSet):  
    separator = ", "  
    return separator.join(map(str, sorted(numberSet)))
```

Putting all the pieces together, we get the program shown as Example 9.1 below.

Notice how the program uses examples of all of the following mathematical and Python constructs:

- streams and generator functions
- 2-tuples
- functional programming with `map`
- sets and set comprehensions
- mappings, dictionaries, and dictionary comprehensions

See how straightforward and concise each part of the program is as a result. The only slightly messy part is the definition of `wordsAndLineNumbers`, but then reading input and taking it apart is often the messiest part of a program. And notice that the whole program is only 19 lines long, not counting blank lines.

Then suppose that the file `document` contains this brief sample of text:

Example 9.1. Finding given words in a document

```
def wordsAndLineNumbers(fileName):
    punctuation = ".,:;" + "'"
    lineno = 0
    file = open(fileName)
    for line in file:
        lineno += 1
        words = line.strip().split()
        for word in words:
            yield (word.strip(punctuation), lineno)

def formatted(numberSet):
    separator = ", "
    return separator.join(map(str, sorted(numberSet)))

targets = { line.strip() for line in open("targets") }
linesFoundOn = { t : set() for t in targets }

for (word,lineno) in wordsAndLineNumbers("document"):
    if word in targets:
        linesFoundOn[word].add(lineno)

for t in targets:
    print(t + " " + formatted(linesFoundOn[t]))
```

Wants pawn term dare worsted ladle gull hoe lift wetter murder inner
ladle cordage honor itch offer lodge, dock, florist. Disk ladle gull
orphan worry putty ladle rat cluck wetter ladle rat hut, an fur disk
raisin pimple colder Ladle Rat Rotten Hut.⁴

And suppose that the file `targets` contains these lines:

```
ladle
gull
hut
```

⁴ Howard L. Chace, *Anguish Languish*, Prentice-Hall, 1956. At the time of writing, available at <http://www.justanyone.com/allanguish.html>, which asserts that the copyright of the book has expired. (Try reading the sample text aloud with a bit of a drawl.)

Then the output of the program might be as follows.

```
hut 3
gull 1, 2
ladle 1, 2, 3
```

The order of the lines of output is unpredictable; if we want them in alphabetical order by word, we could use `sorted` on the set `targets` and write this:

```
for t in sorted(targets):
    print(t + " " + formatted(linesFoundOn[t]))
```

9.4. Dictionary or function?

As we have seen, we can implement a mapping in a program either as a data structure (for example, a static sequence or a dictionary in Python) or as a computation (for example, a Python function). In many respects the two kinds of implementation are equivalent, especially to the program code that uses the mapping, and in Python the notation for using the mapping is similar in the two cases. For example, to get the value to which the mapping `f` maps the value `a`, we write `f[a]` if `f` is a sequence or a dictionary and `f(a)` if `f` is a function.

To choose between a data structure and a computation to implement a particular mapping, a programmer must take into account a number of considerations. Here are some of them:

- Recall that a data structure in a program must be finite in size and also relatively small rather than “finite but far too large” (Section 2.3). A computation may be the only reasonable choice if the domain of the mapping is very large, such as the set of all Python integers or strings.
- Some mappings in programs must be mutable. This is no problem if the mapping is stored as a mutable data structure, such as a list or a dictionary. In most programming languages, program code can't be changed by the program itself, or at least not easily. The mapping implemented by a computation can often be made mutable to some extent, though, if the

computation bases its results on data stored in variables or mutable data structures.

- Once mappings are stored in some data structures such as Python dictionaries, a program can find the value associated with a value in a mapping's domain quite efficiently, often more efficiently than with another kind of computation.
- Depending on circumstances, either the implementation as a data structure or the implementation as a computation may be easier to program. To program a mapping as a computation, a programmer must know a method for computing the mapping, and preferably a reasonably simple method. Some mappings are so apparently arbitrary that they are most easily constructed from explicit data.

But that data must come from somewhere. Small mappings can conveniently be defined using constructs like Python's dictionary displays, so that the data is built into the program. More commonly, the data comes from a source external to the program, such as a file or data base. We have seen how to construct a mapping, as a Python dictionary or sequence, from all the data in a file. On the other hand, sometimes it can be more efficient to program a computation that selectively reads only a small part of a file or data base to implement a mapping, although we will not present techniques for doing so in this book.

Even when there is an obvious computation that defines a mapping, it is sometimes useful to precompute the mapping and store the results in a data structure. Here is an example. Suppose that a Python program has a dictionary-of-dictionaries `distance` that maps pairs of cities to the distance by road (in kilometers, say) between them. Perhaps `distance` was constructed from a flat file, as `population` was in the second construction in Section 9.2. Then, to get the distance between two cities, the program would use an expression like `distance[city][otherCity]`.

Now suppose that the program needs to find a city's nearest neighbor; that is, for a given city, the other city that is a minimum distance away by road. Here is a function that will do that computation:

```
def nearest(city):
    # neighborDistance = a dictionary
    #   { each neighbor of city |-> distance to it }
    neighborDistance = distance[city]

    # until we have at least one candidate:
    nearest = None
    minDistance = float("Infinity")

    # for each neighbor of city:
    for otherCity in neighborDistance:
        if neighborDistance[otherCity] < minDistance:
            nearest = otherCity
            minDistance = neighborDistance[otherCity]

    return nearest
```

The definition of `nearest` illustrates several interesting programming techniques. First, notice that, since `distance` is a dictionary of dictionaries (a curried mapping, so to speak), the function can get the mapping `neighborDistance` by partial application and use it conveniently in the statements that follow. Second, notice that the function returns the value `None` in the (possibly unlikely) event that `distance` defines no neighbors for `city`; it is generally a good idea to program a computation so that it always computes a well-defined value in every situation, however unlikely. Finally, notice how the Python “infinity” value is used. This is a common use for it: as the starting value for a computation that finds a cumulative minimum value.

But notice that `nearest` does a somewhat time-consuming computation. The body of the `for`-statement is executed once for every neighbor of `city`. If each city has an average of n neighbors, a call to `nearest` will consume time approximately proportional to n on average. If the program needs to call `nearest` many times, say m times, the total time will be approximately proportional to mn , which may be prohibitive if m and n are large.

In some situations it may be a good idea to precompute the values of `nearest` for all cities and store the values in a dictionary:

```
nearestStored = { city : nearest(city)
                  for city in keys(distance) }
```

Then a dictionary access `nearestStored[c]` will be much faster than a call `nearest(c)`, especially if n is large. It is true that it takes significant time to construct `nearestStored` — approximately proportional to kn if there are k cities — but if m is much larger than k , the overall savings in time will probably be worthwhile.

It is possible to program a mapping so that it looks up its result in a dictionary in some cases and uses a computation in other cases. There is an interesting programming technique that works this way: it is called *memoization*. No, that's not a typo! The word is “memo-ization” as in keeping a memo.

The idea of memoization is to store the result of a computation so that, if the same computation needs to be done again, the stored result can be used instead. In particular, a memoized function, before returning its result, stores the result in a table using the argument value as a key. If the function is ever called again with the same argument value, the function returns the stored result from the table instead of recomputing the result.

As an example, let's consider the function that calculates a Fibonacci number. In Section 7.2 we saw functions that returned sequences of Fibonacci numbers. Now let's consider the function that, given a positive integer n , returns only the n^{th} Fibonacci number. As we frequently do in programming, we'll count from 0 rather than 1.

Here's a version that calculates the n^{th} Fibonacci number iteratively; that is, by using an iteration. It's a simple adaptation of the function `fib`s from Section 7.2. Notice the simultaneous assignment.

```
def fib(n):
    a = 1
    b = 1
    for i in range(n):
        a, b = b, a + b
    return a
```

This function computes a single Fibonacci number reasonably efficiently, but if a program uses it to compute many Fibonacci numbers, the function may waste a lot of effort. If a program evaluates `fib(1000)`, the body of the for-

statement is executed 1000 times. It would be faster to simply use the sum of `fib(999)` and `fib(998)` if those values happen to be available already.

Let's look at a recursive version of the Fibonacci function. By definition, the first two Fibonacci numbers are 1 and 1, and every other Fibonacci number is the sum of the previous two. Here is a recursive function that is a direct translation of that definition.

```
def rfib(n):
    if n == 0 or n == 1:
        return 1
    else:
        return rfib(n-2) + rfib(n-1)
```

This function works, but it is *very* inefficient. Notice that the last line of the function body computes `rfib(n-2)` and then `rfib(n-1)`, and the latter recursive call ends by computing `rfib(n-2)` again! Similarly, `rfib(n-3)` is re-evaluated unnecessarily (more than once, in fact), and so on.

Now here is a memoized version of the recursive function. It saves its results in a Python dictionary, which is an ideal kind of container for the purpose. The test in the first line of the function body is very efficient, and if it succeeds the function simply returns the previously-stored result.

Example 9.2. A memoized function: the n^{th} Fibonacci number

```
fibStored = {}

def mfib(n):
    if n in fibStored:
        return fibStored[n]
    elif n == 0 or n == 1:
        return 1
    else:
        result = mfib(n-2) + mfib(n-1)
        fibStored[n] = result
        return result
```

All three versions of the function, in the process of computing the n^{th} Fibonacci number, compute all smaller Fibonacci numbers. The iterative function computes each only once. Since the memoized function never recomputes a Fibonacci number that it has computed before (not counting the trivial 0^{th} and 1^{st} , which require essentially no computation), it should be approximately as efficient as the iterative version in computing a single Fibonacci number. Furthermore, in the process, the memoized function fills in the dictionary with the results of computing that number and all the smaller Fibonacci numbers. If a program calls the function again with a different n , the required Fibonacci number may already be in the dictionary, so that the program gets the number essentially for free, computationally speaking; and if not, computing that number fills the dictionary with even more results, increasing the chances that any later call will get its result for free.

9.5. Multisets

We first saw a multiset in Section 1.3, where we saw that the file of temperature observations was a multiset. A multiset (sometimes called a “bag”) has properties in common with a set, but it also has properties in common with a mapping, as we will see in this section.

In mathematics there seems to be no commonly accepted notation for multisets, and multisets are often written with the same curly braces that are used to write sets, so that the multiset that includes 2 twice and 3 once might be written $\{ 2, 2, 3 \}$. In this book we will use different bracketing characters for multisets to distinguish them clearly from sets: we will write the previous multiset as $\llbracket 2, 2, 3 \rrbracket$. This way we can tell unambiguously that (for example) $\{ 2, 3 \}$ is a set but $\llbracket 2, 3 \rrbracket$ is a multiset that happens to contain no element more than once. (The character \llbracket is the Unicode character called “left white curly bracket”, and similarly for the right bracket.)

You may not hear much about multisets in your mathematics classes. In fact, you may hear a lot about sets, but nothing at all about multisets. That's because, to mathematicians, sets are far more important than multisets. Sets pervade all of mathematics, from the foundations on up.

But multisets do have a few important uses in mathematics. For example, the prime factors of a number are a multiset, not a set. Consider the number 360. As you can see by factoring it, 360 is $2 \cdot 2 \cdot 2 \cdot 3 \cdot 3 \cdot 5$, so its prime factors are the multiset $\{ 2, 2, 2, 3, 3, 5 \}$.

To take another example, the roots of an algebraic equation can be viewed as a multiset. Recall some simple algebra: the equation $x^2 - 5x + 6 = 0$ can be written as $(x - 2)(x - 3) = 0$, so its roots are 2 and 3. In one way of defining roots, a quadratic equation always has two roots, but they may be the same. For example, the equation $x^2 - 4x + 4 = 0$ can be written as $(x - 2)(x - 2) = 0$, so its roots are the multiset $\{ 2, 2 \}$.

Consider the collection of different items available in a particular shop. Clearly, that's a set. But what about the collection of all the items on the shelves — the inventory? It's a multiset. In taking an inventory, an obvious way to proceed is to take the set of different items and count how many of each item there are.

In fact, one way of looking at a multiset is as a mapping from elements to the count of each. For example, the multiset $\{ 2, 2, 2, 3, 3, 5 \}$ can be treated as equivalent to the mapping $\{ 2 \mapsto 3, 3 \mapsto 2, 5 \mapsto 1 \}$. Then we can get the number of occurrences of a value in a multiset, if it occurs there at all, using functional notation; for example, if the previous multiset is m , then the count of 3 in m is $m(3)$ or 2.

Let us define a function *count*, which gives the number of occurrences of a value in a multiset, as follows: for a multiset m and a value a , $count(m,a)$ is $m(a)$ if $a \in \mathbf{dom} m$ and 0 otherwise.⁵

Using *count* we can define other operations on multisets. For example, the multiset sum operator, written \uplus , takes two multisets as operands and produces another in which the counts of elements are the sums of the counts in the two operands. Multiset union is different: every count is the maximum of the counts from the operands, rather than the sum. Multiset intersection uses

⁵In mathematics the term “multiplicity” is often used instead of “count”, but we will use the shorter word.

minima instead of maxima. In other words, the three operators are defined by these equations: for multisets m and n and element a ,

$$\text{count}(m \uplus n, a) = \text{count}(m, a) + \text{count}(n, a)$$

$$\text{count}(m \cup n, a) = \max(\text{count}(m, a), \text{count}(n, a))$$

$$\text{count}(m \cap n, a) = \min(\text{count}(m, a), \text{count}(n, a))$$

There are two obvious implementations of a multiset in Python: either as a sequence of values (a tuple or a list, depending on whether the multiset needs to be mutable), or as a dictionary mapping values to counts. Either implementation has its advantages and disadvantages.

In the implementation as a sequence, the elements have an order but the order is not important; we use a sequence rather than a set only because a sequence can contain the same value more than once. The built-in Python method `count` is exactly the function `count`: recall that, if m is a sequence, `m.count(a)` gives the number of occurrences of a in m . Python must traverse the whole sequence to do the counting, though, so the operation can be time-consuming if m is large. Multiset sum can be simply concatenation. Multiset union and intersection, on the other hand, require more computation: see Exercise 14.

The implementation as a dictionary is probably better if the most common operation will be to access counts of elements, as is frequently the case in programs. Where it is certain that a is in multiset m , the count is simply `m[a]`, which Python can evaluate quite efficiently. In the general case, `count` must be programmed as a function, but the function is simple and again Python can compute it efficiently:

```
def count(m,a):
    if a in m:
        return m[a]
    else:
        return 0
```

Multiset sum, union, and intersection require a bit more computation: see Exercise 15.

It is easy to construct a multiset of the lines of a flat file. For a tuple implementation, `tupleOfValues` from Section 8.4 gives us the multiset directly, and similarly for a list implementation. If we want to construct a dictionary implementation, we can use `streamOfValues` from the same section as follows:

```
def multisetOfValues(fileName):
    m = {}
    for a in streamOfValues(fileName):
        if a in m:
            m[a] += 1
        else:
            m[a] = 1
    return m
```

Many other multiset operations that you are likely to need in a Python program will be simple variations on that code. For example, suppose you want to treat the lines of a file as integer values, as in the program of Example 1.3. The following function constructs a multiset of those values:

```
def multisetOfIntegers(fileName):
    m = {}
    for a in streamOfValues(fileName):
        n = int(a)
        if n in m:
            m[n] += 1
        else:
            m[n] = 1
    return m
```

You can probably see easily how the function would look if you wanted to construct a multiset of values from a file transformed in some other way, or a multiset of the values of a particular field in the file lines, and so on.

The if-statement in those functions is a common pattern of code and, packaged as a function on its own, can be useful in many programs:

```
# tally(m,x):
#   add 1 to the count of value x in multiset m

def tally(m,x):
    if x in m:
        m[x] += 1
    else:
        m[x] = 1
```

In the common situation in which a program uses a multiset as a set of counters, the two central operations on the multiset will often be the functions `tally` and `count` that we have just defined.

Here's a simple example of a program that uses a multiset as a set of counters. Given the file `students` that we used in Section 8.4, the program tabulates and displays the number of students in each major field of study represented in the file. Using the function `tally` and the function `streamOfTuples` from Example 8.2, it's as easy as this:

Example 9.3. Number of students in each major field

```
counts = {}

for (name,major) in streamOfTuples("students"):
    tally(counts,major)      # use major, ignore name

for major in counts:
    print major + ": " + counts[major]
```

Terms introduced in this chapter

maplet	powers of a mapping
domain	dictionary display
range	dictionary comprehension
curried function	key
domain restriction	memoization
overriding union, override	multiset sum, union, intersection
composition	

Exercises

1. Consider the following Python function:

```
def p(x,y):
    return x*y
```

Python gives us no way to say that we mean the domain of `p` to be pairs of integers, or pairs of floating-point numbers, or something else. As it stands, what is the domain of `p`?

2. Convince yourself, using examples, that the \oplus operator is associative. Then show why $(f \oplus g) \oplus h = f \oplus (g \oplus h)$ for any mappings f , g , and h . Use pictures if you like; or, if you can, give a mathematical proof.
3. Consider mappings c and d . Under what conditions on c and d does $c \oplus d = d \oplus c$?
4. Write a higher-order Python function that returns the composition of two one-argument functions, as a one-argument function.
5. Write a Python function that returns the composition of two dictionaries, as a dictionary.
6. If f is a function, does the notation f^0 have a sensible meaning? If so, exactly what does f^0 denote?
7. When we defined f^3 as $f \circ f \circ f$ we did not use parentheses in the latter expression. Was this justified? Is composition of functions associative? Does it have an identity? If so, what is it? Do we have another monoid here?
8. Which analog of a two-dimensional array — the single dictionary using ordered pairs as keys, or the dictionary of dictionaries — is preferable in Python? Which representation can the Python interpreter operate on more efficiently? Conduct an experiment: construct two structures, one of each kind. Then time how long it takes the Python interpreter to access all elements of each structure (to set all the elements to zero, for example).

Discuss what considerations besides speed you might use to decide which representation to use in a particular program.

9. Look again at the function `wordsAndLineNumbers` in the program of Section 9.3. Did it occur to you that this code zips together two sequences

(Section 7.5)? Rewrite `wordsAndLineNumbers` using `zip`. Do you like this version of the program better?

10. Modify the program of Section 9.3 so that it finds words in the document ignoring the distinction between upper and lower case letters. For example, with the sample files given, the program would match “hut” in the file `targets` not only with “hut” on line 3 of the file `document` but also with “Hut” on line 4. (Hint: look in the Python documentation for a function that will help.)
11. In the `nearest` function of Section 9.4 we used `float("Infinity")` as the starting value in a cumulative-minimum computation. Is it only a coincidence that `float("Infinity")` is the identity of the monoid of Python floating-point numbers under the minimum-function (Section 6.2)? What would you use as the starting value in a cumulative-maximum computation? Can you think of similar computations that you might do in other monoids? How are such computations related to “big” operators and to the `reduce` function of Section 6.4? Can you rewrite `nearest` using `reduce` instead of an iteration?
12. Experiment with versions of the `fib` function. Is the memoized recursive version more efficient than the iterative version? Run each version with a sequence of calls, using a variety of argument values, to give the memoized version a chance to use its stored results.

You should expect that the non-memoized recursive version of the function will be much slower than the other two versions for large values of `n`. Experiment to see how large `n` needs to be to make the non-memoized recursive version impossibly slow.

13. Memoize some other functions and experiment with them to see whether the memoized versions are more efficient than the usual versions. You might try `nearest` (Section 9.4), `factorial` (Section 4.2), or `isPrime` (Exercise 2 of Chapter 8). Again, run each version of each function with a sequence of calls using a variety of argument values.

14. Write Python functions for multiset union and intersection, assuming that multisets are implemented as lists.
15. Write Python functions for multiset sum, union, and intersection, assuming that multisets are implemented as dictionaries. In the dictionaries that the functions return, don't include any keys that map to zero.

Chapter 10

Relations

10.1. Mathematical terminology and notation

We first saw relations in Section 1.3, where we described relations that are sets of ordered pairs. As we noted in Section 5.3, though, relations in mathematics are more generally sets of n -tuples for any n .

A relation that is a set of ordered pairs is called a binary relation, as we have seen. There are names for relations that are sets of n -tuples for larger n ; for example, a set of 3-tuples is a ternary relationship and a set of 4-tuples is a quaternary relation. The general term is *n -ary relation*. As with n -tuples, we can substitute a specific number for n , so that (for example) a ternary relation is a 3-ary relation. The file `populations` of Section 9.2 was conceptually a 3-ary relation: a set of triples in which each triple contained a city, another city, and the distance between them.

If an n -tuple is in a relation, we often interpret that fact as implying that some corresponding relationship holds among the elements of the n -tuple. For example, consider the relation defined by the data in the file `emails` of Example 1.2. Call that relation E ; then $(x, e) \in E$ means that person x has email address e .

In mathematics a binary relation is often used as a binary operator: for example, if $(x, e) \in E$, we can also write $x E e$. Conversely, a binary operator that yields a value of true or false — that is, a Boolean operator — defines a binary relation. For example, the mathematical “=” operator defines a relation that contains (a, a) for every a in the universe of discourse. (If that universe is infinite, then the relation is an infinite set, which illustrates that relations can be infinite.) We can even use the operator as the name of the relation, and write $(a, a) \in =$.

More generally, an n -ary relation defines a *predicate*, a statement about n things that is true if the n -tuple of those things is in the relation and false otherwise. Or, conversely, such a statement defines a relation. The notation of functions is often used for predicates: for example, if P is a 3-ary relation and $(a, b, c) \in P$, then $P(a, b, c)$ has the value true. Similarly, if $(a, b, c) \notin P$, then $P(a, b, c)$ has the value false.

Often we use a longer name, rather than a single letter or symbol, for a relation or an operator or a predicate; as we have seen, mathematicians and computer scientists can be rather free in their use of notation. For example, we may write “*equals*” instead of “=”, or “*hasEmail*” for the relation of Example 1.2. And we often use the notations of relations and operators and predicates interchangeably, so that (for example) in each row of the table below all of the expressions have the same meaning.

relation	operator	predicate
$(x, e) \in E$	$x E e$	$E(x, e)$
$(a, b) \in =$	$a = b$	$=(a, b)$
$(a, b, c) \in P$		$P(a, b, c)$
$(a, b) \in \textit{equals}$	$a \textit{ equals } b$	$\textit{equals}(a, b)$
$(x, e) \in \textit{hasEmail}$	$x \textit{ hasEmail } e$	$\textit{hasEmail}(x, e)$

A 1-ary relation (a set of one-element tuples) is seldom very useful, but predicates of one argument are common in mathematics. We usually interpret such a predicate as a property that a single thing may or may not have, and then the predicate defines a set rather than a relation: a one-argument predicate P defines the set of all a for which $P(a)$ is true. To take an example from Section 8.1, suppose that $\textit{smallOdd}(n)$ represents “ $0 < n < 100$ and n is odd”. Then the predicate $\textit{smallOdd}$ defines the set of numbers with that property; that is, $\{n \mid \textit{smallOdd}(n)\}$.

When a relation R consists of n -tuples whose elements are all members of the same set A , we say that R is a *relation on A* . The set is an important property of the relation. For example, many binary relations that are important in mathematics and computer science are denoted by binary operators, but if an

operator is overloaded we must know the set that the relation is on to know what relation we are talking about. The “ $<$ ” relation on integers is not the same as the “ $<$ ” relation on real numbers, because they are completely different sets of ordered pairs.

Notice that any relation is always on *some* set, whether the set is explicitly given or not; the smallest set satisfying the definition of “on” is implicitly defined by the n -tuples in the relation. If the relation is given but the set is not, and if the relation is finite and reasonably small, we can compute the set from the relation.

Mathematicians have identified a number of properties that some binary relations have, and we'll mention a few of them here. A binary relation R is *symmetric* if, whenever a pair (a, b) is in R , then (b, a) is also in R . For example, the relation “ $=$ ” on integers is symmetric, because if $a = b$, then $b = a$.

On the other hand, a binary relation R is *antisymmetric* if, whenever a pair (a, b) is in R , then (b, a) is not in R . For example, the relation “ $<$ ” on integers is antisymmetric. Notice that from the fact that (b, a) is not in R we can't conclude that (a, b) is in R ; consider the relation “ $<$ ” and the pair $(3, 3)$, for example.

A binary relation R is *transitive* if, whenever $(a, b) \in R$ and $(b, c) \in R$, then $(a, c) \in R$. The relations “ $=$ ” and “ $<$ ” on integers are both transitive.

If R is a relation on a set A , then we say that R is *reflexive* if $(a, a) \in R$ for all $a \in A$. For example, “ $=$ ” on integers is reflexive, but “ $<$ ” is not.

A binary relation that is symmetric, transitive, and reflexive is called an *equivalence relation*. For example, “ $=$ ” on integers is an equivalence relation. So is “born in the same year” on pairs of people.

It is not hard to show that an equivalence relation R on a set A defines a *partition* on A ; that is, a set of subsets of A such that every element of A is in some subset and no element is in more than one. The relation R partitions A into subsets of elements that R defines as being equivalent to each other.

The relation consisting of only (a, a) for all $a \in A$ is called the *identity relation* on A , and is sometimes written I_A . Notice that this relation is a mapping; it can also be called the *identity mapping* or *identity function* on A .

Composition of binary relations is defined like composition of mappings (Section 9.1): for two relations R and S ,

$$S \circ R = \{ (a, c) \mid (a, b) \in R \text{ and } (b, c) \in S \}$$

Powers of binary relations are also defined as with mappings: for a relation R , we write R^2 for $R \circ R$, we write R^3 for $R \circ R \circ R$, and so on. We define R^0 to be the set of all (a, a) and (b, b) for all $(a, b) \in R$; that is, R^0 is I_A for the smallest A on which R is defined. We define R^1 in the obvious way: it is just R .

10.2. Representations in programs

Programs frequently represent and compute with relations of one kind or another. In implementing a relation, a programmer faces many of the same choices as in implementing a mapping or a multiset. Should the representation be as a data structure, a computation, or some combination of those? If a data structure, what kind?

In Python, the most obvious representation of a relation is just what the definition of “relation” implies: a set of tuples. This representation also has the advantage of making some operations particularly convenient, such as iterating through the tuples in the relation or finding whether a given tuple of values is in the relation.

In some situations another representation may be more convenient, though. For example, consider a binary relation containing pairs (a, b) . If the most common operation that a program will perform is to find the b values associated with a given a value, it might be convenient to represent the relation as a mapping that maps an a value to a set of b values. On the other hand, this representation makes it rather inconvenient to find the a values associated with a given b value, should that operation be necessary too.

Sometimes a computation is a better representation than a data structure for a relation, just as for a mapping and for any of the same reasons (Section 9.4). There are two obvious kinds of implementation as a computation:

- As code that enumerates the tuples in the relation: in Python, a generator function or generator expression, for example. In some situations a stream or other sequence of all the tuples is just what a program needs. And occasionally, even if the enumeration would be infinite or very large, a program might need only some of the tuples of the relation and could take only a prefix of the sequence. Most often, though, the implementation as an enumeration is less useful than the alternative, which is:
- As a predicate; that is, as a function that returns a Boolean value. Such a function takes n arguments or a tuple of n values and returns `True` or `False` according to whether the corresponding n -tuple is in the relation. The function can do any computation necessary to determine whether it is.

As with a mapping, it can be advantageous to program a predicate partly by accessing a data structure and partly by a computation. For example, the computation can be memoized.

Or we can often make use of special properties of relations, as defined in the previous section, to infer that some tuples are in a relation even though they are not stored in the data structure. For example, suppose that a relation r is symmetric. We can construct a Python set `rStored` (perhaps from data in a file) that contains pairs (a, b) in one direction, but not the corresponding pairs (b, a) in the other direction. Then we can implement the predicate defining r by augmenting `rStored` with program logic as follows:

```
# predicate r(a,b):
#   is the pair (a,b) in the symmetric relation r?

def r(a, b):
    if (a,b) in rStored:
        return True
    elif (b,a) in rStored:
        return True
    else:
        return False
```

or, more briefly,

```
# predicate r(a,b):  
#   is the pair (a,b) in the symmetric relation r?  
  
def r(a,b):  
    return (a,b) in rStored or (b,a) in rStored
```

This technique cuts in half the quantity of stored data we need. Similarly, if a relation is reflexive, there may be no need to store all the pairs (a,a) for all the a in the set on which the relation is defined. In situations in which we will only invoke the predicate for pairs of values that are in that set, we can implement the predicate like this:

```
# predicate r(a,b):  
#   is the pair (a,b) in the reflexive relation r?  
  
def r(a,b):  
    return a == b or (a,b) in rStored
```

We can use similar techniques with relations that are not binary, but that are symmetric or reflexive in some other sense. For example, consider the examples in Section 9.4 in which we computed with distances between cities. We can treat the data in those examples as a 3-ary relation whose triples are two cities and the distance between them. Since most roads between cities go in both directions, the road distance between any two cities is probably the same in both directions, so the relation is symmetric in that sense. And we can probably treat the distance from every city to itself as zero, as a kind of reflexive property.

Under those assumptions, we can construct a dictionary-of-dictionaries `distanceStored` that contains only distances between different cities in one direction, and then write a distance function as shown in Example 10.1 below. With this implementation we decrease the amount of data that needs to be stored and possibly the work required to generate the data, at the expense of increased computation time and increased complexity of the code. Tradeoffs like this one are common in software design.

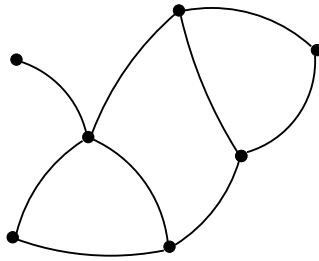
Example 10.1. Distances using symmetry and reflexivity

```
def distance(city,otherCity):
    if city == otherCity:
        return 0.0
    elif city in distanceStored and \
         otherCity in distanceStored[city]:
        return distanceStored[city][otherCity]
    elif otherCity in distanceStored and \
         city in distanceStored[otherCity]:
        return distanceStored[otherCity][city]
    else
        return float("Infinity")
        # no road connection that we have data for
```

Transitive relations are harder to deal with, and trying to do something similar for a transitive relation doesn't work. Suppose we want to write a predicate *r* for a transitive relation based on minimal data in a set of pairs *rStored*. We would need to return True for a pair (a,c) if there are pairs (a,b) and (b,c) in *rStored* for some b, but how do we find the b? Even worse, we would need to return True for a pair (a,d) if there are pairs (a,b) and (b,c) and (c,d) in *rStored* for some b and c, and so on for any longer such chain of values that might exist in *rStored*. We'll see how to overcome these difficulties in Section 10.4.

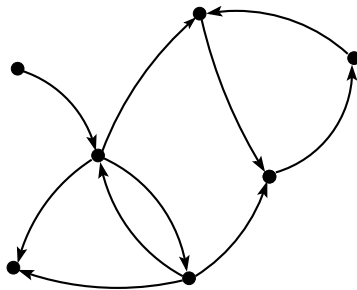
10.3. Graphs

In mathematics, a *graph* is a diagram showing points or other objects and links connecting them, or a mathematical object that represents such a diagram. In common English usage we think of a “graph” as a plot of data such as the one in Figure 1.1 (page 15), but in mathematics the term has a specialized meaning. Here is an example of a graph, shown as a diagram.



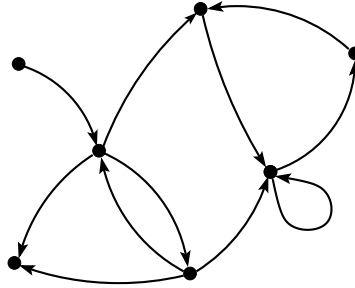
As a mathematical object, a graph is usually defined as an ordered pair (V, E) , where V is a set of *vertices* (the points or objects; the singular is “vertex”) and E is a collection of *edges* (the links). Starting with this basic definition, there are many variations depending on how V and E are defined. Different authors use different definitions and terminology, and different definitions define kinds of graphs with different properties.¹ All the definitions that we will see here use the various structures of discrete mathematics that we have been discussing throughout the book, and it is interesting to see how these structures can be combined in different ways according to what one wants to express in a definition.

In an *undirected* graph, an edge has no direction and is simply like a line connecting two vertices, as in the example above. In a *directed* graph, each edge has a direction and is like an arrow from one vertex to another, as in the following example. Edges in a directed graph are sometimes called “directed edges” or “arcs”.



¹ As always, whatever you're reading, read the definitions carefully!

In either kind of graph, a *loop* is an edge connecting a vertex with itself. The previous two graphs have no loops, but the following graph has one.



In one definition of a directed graph, E is a set of ordered pairs of vertices (elements of V). In one definition of an undirected graph, E is a set of two-element sets of vertices, since those vertices have no order.

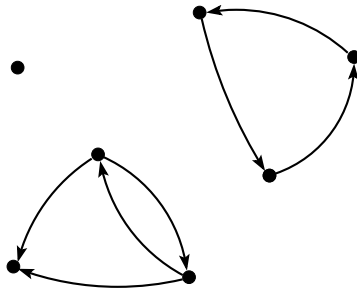
But notice that the latter definition does not allow undirected graphs to contain loops, since a set cannot contain two identical elements. If we want the definition to allow loops, the edges must be two-element multisets rather than sets.

As described until now, an edge in a graph simply represents the fact that two vertices are connected. Sometimes, though, a graph must represent the fact that there is more than one connection between a pair of vertices (we call these *multiple edges*). Consider a graph that represents the Internet, for example. The vertices will probably be computers or computing sites, and the edges (probably undirected) will be physical or virtual communication links. In some places the Internet has multiple links between pairs of sites for increased capacity or reliability. A graph that is used to help with routing of messages or packets probably needs to contain multiple edges to represent the multiple links.

Definitions of a graph (directed or undirected) that make E a set of edges do not allow for multiple edges. One solution is to define E as a multiset, not a set, of pairs or sets or multisets of vertices.

Some mathematicians take another approach and define a directed graph as a 4-tuple $(V, E, head, tail)$. Here V and E are simply sets of abstract objects, and $head$ and $tail$ are mappings from edges to vertices. An undirected graph has a single mapping from edges to two-element sets or multisets of vertices. Such definitions allow multiple edges, because any two elements of E are different by definition even though they may have the same ends.

Notice that all of these definitions allow a graph to contain subsets of vertices with no edges connecting the subsets with each other; there is nothing in any of the definitions that says otherwise. In fact, there may even be vertices that are isolated, connected to no other part of the graph.



Unfortunately, the literature of mathematics and computing is not at all consistent in definitions and terminology regarding graphs. In many mathematics textbooks the term “graph” means an undirected graph without loops or multiple edges; where the terms “directed graph” and “undirected graph” are used, they often still refer to graphs having no loops or multiple edges. The books may not say so explicitly, but the definitions in the books, like some of the definitions above, imply these facts. In such books, graphs that may contain loops or multiple edges or both, if they are discussed at all, are given some other name such as “multigraph”.

On the other hand, the term “graph” may be defined so as to allow loops and multiple edges. If so, the term “simple graph” is often used for a graph with neither.

This confusion of definitions and terminology has sometimes led programmers and computer scientists to make mistakes in their algorithms. For example, a

programmer may learn a definition of “graph” that is really only appropriate for simple graphs, and then rely on that definition in constructing a graph algorithm that should allow for loops or multiple edges but does not. Beware of such mistakes!

There is an intimate relationship between graphs and binary relations; in fact, we can often view a graph and a binary relation as two different ways of representing the same information. A directed graph (V, E) defines a binary relation on V , sometimes called the *adjacency relation* of the graph, which contains the ordered pair (a, b) if E contains an edge (a, b) . Conversely, a binary relation defines a directed graph, sometimes called the *graph of the relation*, that contains an edge (a, b) if the relation contains the ordered pair (a, b) . An undirected graph defines a symmetric relation, and vice versa, similarly. But notice that none of these relations can represent any multiple edges that may be present in the corresponding graph.

In a program, the edge set of a directed graph without multiple edges can be represented in any of the ways that a binary relation can, as described in the previous section, and the same design considerations apply. If the graph contains multiple edges, similar representations are possible where a multiset of ordered pairs replaces the binary relation. If the graph is undirected, the representation can be like that of a symmetric directed graph, with a directed edge in each direction for each edge in the undirected graph. Other representations are sometimes useful to facilitate particular computations on graphs, such as the computations that we will consider in Section 10.4, and we will see a quite different representation in Section 11.6.

Graphs can be augmented to contain additional information. For example, a *labeled graph* is a graph in which the vertices or edges or both have identifying labels, such as names or numbers. For example, in a graph representing a road map, the vertices might be labeled with names of cities and the edges might be labeled with names of roads.

In a directed graph with labeled edges, an edge can be defined as a triple of two vertices and a label. If every edge has a different label, the problem of multiple edges largely goes away: the collection E of edges can again be defined

as a set, and multiple edges between two vertices are distinct members of the set, distinguished by their labels. Similar comments apply to undirected graphs with labeled edges. In a graph with labeled vertices in which every vertex label is different, the label of each vertex can sometimes be a convenient representation of the vertex in a program.

A *weighted graph* is a graph in which each edge is associated with a number, called its “weight”. The meaning of the weight depends on what the graph represents. For example, in a weighted graph in which the vertices represent cities, the weight of an edge may represent the length of a road between two cities, as in the examples of Section 9.4, or the time needed to travel between two cities by train or air. The dictionary-of-dictionaries in Example 10.1 shows one way to define a weighted graph and represent it in a program: essentially, using a mapping from edges to weights. In that example the edges are pairs of vertices; apparently there are no multiple edges. If that is the case, another representation of the graph could be as a set of triples, each triple being two vertices and a weight. But in the presence of multiple edges this representation would not work, because usually there is no reason to believe that all the weights are different.

A graph can be both labeled and weighted. Graphs that are labeled or weighted or both are common in computational problems.

10.4. Paths and transitive closure

In computing, one of the most common operations on graphs (directed or undirected) is to find *paths* from one vertex to another. A path in a graph can be defined as a sequence of vertices such that there is an edge in the graph from each vertex to the next, and a path between two vertices a and b can be defined as a path beginning at a and ending at b . Alternatively, a path can be defined as the sequence of edges, and this is the definition that we will usually use. We will also say that there is a path of length zero, the empty sequence of edges, from every vertex to itself. Not all mathematicians would allow empty paths, but if we allow empty sequences it makes sense to allow empty paths, and the generalization can be useful for avoiding special cases in computer algorithms.

Many programs need to compute answers to questions such as these:

- For a given graph and vertices a and b in it, is there a path from a to b ?
- If so, what is the length of the shortest path, meaning the path with the fewest edges?
- Or, if the graph is weighted, what is the length of the minimum-weight path, meaning the path with the smallest sum of weights?
- Does the graph contain a *cycle*, meaning a nonempty path from a vertex to itself? Does the graph contain a cycle that includes some given vertex?
- If the graph is labeled, what is a path (if any) that answers each of the above questions, as a sequence of labels?
- Is the graph *connected*: is there a path from each vertex to every other vertex?

Some obvious examples of programs that do computations like these are the popular programs that give driving directions for the fastest route between two cities, and programs that route messages or packets from one site to another in the Internet.

We can gain insight into the questions above by considering the adjacency relation of a graph instead of the graph itself, and by considering the *transitive closure* of that relation. The transitive closure of a relation R is defined as the smallest relation containing R which is transitive; that is, the relation we get by adding to R all the ordered pairs implied by the transitive property and no others. The transitive closure of R is often denoted as R^+ .

One way to find the transitive closure of a relation R is to build it up iteratively. We start with R . Then we find R^2 : for every pair (a, b) in R , we find every pair (b, c) , and the set of pairs (a, c) is R^2 . We add that set to our result; we now have $R \cup R^2$. Now we find R^3 by composing R and R^2 similarly, and add R^3 to our result. We continue in this way; then finally we get R^+ , which is $R \cup R^2 \cup R^3 \cup \dots$

If R is a finite relation (that is, a finite set of pairs), we do not need to compute an infinite union to find R^+ : eventually no new pairs will be added to the result, and we can stop. The set of all a and b such that $(a, b) \in R$ is a finite set, and there are only finitely many possible pairs of members of that set. Therefore, we will eventually reach a k such that every pair in R^{k+1} is already in $R \cup R^2 \cup R^3 \cup \dots \cup R^k$. We can terminate the iterative building of R^+ when this is the case. Thus the method described above is a sketch of an algorithm for computing R^+ , guaranteed to terminate if R is finite. We will call it the Obvious Algorithm for transitive closure.

Now notice that if R is the adjacency relation of G , pairs in R^2 are end points of paths of length 2 in G , pairs in R^3 are end points of paths of length 3 in G , and so on. Then R^+ is the set of pairs of vertices (a, b) such that there is a path of any nonzero length from a to b in G .

So if G is finite, we can compute R^+ , and it contains the information we need to answer many of the questions about paths that we listed above. Many variations on the Obvious Algorithm are possible depending on the question we want to answer. For example, if we are only interested in paths starting at a given vertex a , we don't need to compute the whole relation R^+ ; we can start with only the relation $\{ (a, b) \mid (a, b) \in R \text{ for some } b \}$ and repeatedly compose that relation with R as in the Obvious Algorithm. For questions involving labels or weights, we can modify the Obvious Algorithm to keep track of the labels and weights associated with each path we find.

The Obvious Algorithm is not the most efficient algorithm for computing a transitive closure in all situations or for all kinds of relations or graphs, and it is not really practical at all for large relations or graphs. Computer scientists have studied transitive closures and paths in graphs extensively, and there is a substantial literature describing improved algorithms, as well as data structures that make the computations easier. All of these results are beyond the scope of this book, but if you are a computer science student you will probably see some of them later in your studies. And if you ever need to write a serious program involving transitive closures or paths in graphs, especially if the amount of data is large, you should probably investigate the literature before getting too far into designing the program.

Other closures are possible besides the transitive closure. For example, the symmetric closure of a relation R is formed by adding to R the pair (b, a) for every $(a, b) \in R$. The reflexive closure of a relation R is formed by adding to R the pairs (a, a) and (b, b) for every $(a, b) \in R$.

The reflexive transitive closure of a relation R is the reflexive closure of R^+ ; it is often denoted as R^* . Then $R^* = R^0 \cup R^1 \cup R^2 \cup R^3 \cup \dots$. If R is the adjacency relation of a graph G , there is a pair (a, b) in R^* whenever there is a path, empty or not, from a to b in G .

Transitive closures and reflexive transitive closures have many uses in mathematics and computer science besides finding paths in graphs but, again, those uses are outside the scope of this book. The main purpose of the current section is to introduce you to concepts so that you will start thinking along the right lines whenever you see a programming problem that can be solved by finding closures or paths in graphs.

By the way, the superscript “+” and “*” are common idioms in computer science and mathematics, and are used to mean “one or more times” and “zero or more times”, respectively, in all kinds of contexts. A computer science student soon learns that a superscript “*” is not necessarily an instruction to look for a footnote at the bottom of the page!

10.5. Relational database operations

Now that we have looked at relations in a bit more detail, let us return to the topic of relational databases.

As we mentioned in Section 5.3, a relation in a relational database is very much like a mathematical relation, being a set of “tuples”. However, in a relational database tuple, the elements are identified by name rather than by position.

In the terminology of relational databases, one property of a relation is the set of *attributes* of its tuples. Each attribute has a name, and a tuple is a set of named attribute values.

We can describe a database relation in terms of a mathematical relation plus some additional information about attribute names. Let us represent a database relation with n attributes as a triple (R, A, col) , where R is an n -ary mathematical relation, A is a set of n attribute names, and col (for “column”) is a mapping from attribute names to positions, which are integers in the range 0 to $n-1$. For a particular database relation, say X , we can call the components of the representation R_X , A_X , and col_X .

In the representation, some order is chosen for the attributes, and the tuples in R all have their attribute values in that order. Then R is like a table, with tuples corresponding to rows and attributes corresponding to columns. The order of the attributes defines the mapping col .

In a small and simple relational database, relations could be implemented much as the above description would suggest. For example, the R of a relation could be represented in a program as a set of tuples, and in a file system as a flat file. However, the implementation of a real, production-quality relational database system is usually much more complicated, with additional data structures and algorithms to make access and operations more efficient. All this complexity is, once again, beyond the scope of the current book. Here we'll just use our representation (R, A, col) to talk abstractly about database relations and operations on them, and to help us picture what is going on.

Relational database operations are designed to allow users, often managers and information technology staff in organizations, to combine and extract information contained in relations in a database. We'll describe a few of the most important relational database operations informally here.

Relation union, intersection, and difference are defined much like the corresponding operations on sets of tuples. These operations on two relations X and Y make sense only if the relations have the same attributes; that is, we must have $A_X = A_Y$.

Let us picture Y' as the relation we would get by reordering the “columns” of Y , if necessary, so that the attributes are in the same order as in X . Thus $A_{Y'} = A_X$, $col_{Y'} = col_X$, and $R_{Y'}$ is R_Y with the values in each tuple reordered to agree with col_X .

Then the union of relations X and Y is another relation Z , defined as follows: $R_Z = R_X \cup R_Y$, $A_Z = A_X$, and $col_Z = col_X$. The intersection and difference of X and Y are defined similarly.

In implementing these operations, we could just as well choose to reorder the first relation rather than the second. But the implementations would be simpler if no reordering of either relation would ever be necessary. To ensure that this is the case, we could define a standard ordering of attributes for all relations in the database: alphabetical by attribute name, for example.

A very common database operation is *selection*: to select from a relation all tuples that meet some criterion. This operation is very much like the `filter` function that we first saw in Section 6.4, and like filtering using “`if`” in Python comprehensions. In mathematical notation a selection is written as $\sigma_P(X)$, where X is a relation and P is a predicate or Boolean-valued expression (here σ is a lower-case Greek sigma). The result is a new relation. For example, consider this relation from Section 5.3:

name	project	lab
Lambert	Alpha	221
Torres	Alpha	244
Malone	Beta	152
Harris	Beta	152
Torres	Beta	152

If this relation is X , and if P is “project = ‘Alpha’”, then $\sigma_P(X)$ is

name	project	lab
Lambert	Alpha	221
Torres	Alpha	244

Often a database manipulation needs only some of the “columns” of a relation. A *projection* of a relation is a new relation containing a subset of the attributes of the original relation. In mathematical notation, a projection is written as $\pi_S(X)$, where X is a relation and S is a set of names of attributes to be included

in the result (here π is a lower-case Greek pi). For example, if X is as above and S is { “name”, “lab” }, then $\pi_S(X)$ is

name	lab
Lambert	221
Torres	244
Malone	152
Harris	152
Torres	152

There are a number of operators for combining information from more than one relation. The most important of these is the *natural join*. If Y and Z are relations, their natural join, written $Y \bowtie Z$, is created from pairs of tuples in Y and Z that agree in values of the attributes that the two relations have in common. The resulting relation contains tuples whose attributes are the union of the attributes of the pairs of tuples from Y and Z . For example, let X be as above and let W be this relation:

lab	equipment
221	400913-C
152	400697-A
152	19472832

Then $X \bowtie W$ is

name	project	lab	equipment
Lambert	Alpha	221	400913-C
Malone	Beta	152	400697-A
Malone	Beta	152	19472832
Harris	Beta	152	400697-A
Harris	Beta	152	19472832
Torres	Beta	152	400697-A
Torres	Beta	152	19472832

By the way, the operator \bowtie is sometimes called the “bowtie operator”.

Using natural join and projection it is possible to perform an operation that is much like a mathematical composition of relations. For example, consider the two-attribute database relations $\pi_S(X)$ and W above: they are much like mathematical binary relations. Combining them with a natural join, we obtain

name	lab	equipment
Lambert	221	400913-C
Malone	152	400697-A
Malone	152	19472832
Harris	152	400697-A
Harris	152	19472832
Torres	152	400697-A
Torres	152	19472832

Then we perform a projection to remove the common attribute, and we obtain

name	equipment
Lambert	400913-C
Malone	400697-A
Malone	19472832
Harris	400697-A
Harris	19472832
Torres	400697-A
Torres	19472832

This result is the relational database equivalent of $W \circ \pi_S(X)$.

Terms introduced in this chapter

n -ary relation
 predicate
 relation on a set

loop
 multiple edges
 adjacency relation

symmetric relation	graph of a relation
antisymmetric relation	labeled graph
transitive relation	weighted graph
reflexive relation	path
equivalence relation	transitive closure
partition	symmetric closure
identity relation	reflexive closure
identity mapping/function	reflexive transitive closure
relation composition	attribute
power of a relation	relational union, intersection, difference
graph	selection
vertex	projection
edge	natural join
directed graph	
undirected graph	

Exercises

1. Obviously I_A is reflexive for any set A . Is it transitive? Is it an equivalence relation?
2. In Section 10.3 we gave a very informal definition of the “graph of a relation”, but in that definition we described only the set of edges of the graph. What is the set of vertices? Is there more than one possible choice?
3. Write a Python function that returns the transitive closure of a relation, where the relation and the result are Python sets of two-element tuples.
4. If G is a finite graph and R is its adjacency relation, and if you compute R^+ using the Obvious Algorithm, what is k , the number of powers of R that the algorithm will compute before terminating? Can you find an approximate value, an upper bound, or even an exact value? Express your answer in terms of paths in G . This k will usually be much less than the number implied by the argument that there are only finitely many pairs that R^+ can contain.

5. In a given directed graph, if the vertex at the end of one path is the same as the vertex at the beginning of another path, the two paths can be concatenated to form another path in the graph. Is the set of all paths in a given directed graph a monoid under concatenation?
6. What is $\sigma_P(X)$ expressed as a triple? In other words, what are its R , A and col ? Assume that you can use P as a predicate, writing expressions like “ $P(x)$ ”.

Repeat for $\pi_S(X)$ and $X \bowtie Y$. These are a bit harder, at least if you do them entirely in mathematical notation.

7. What are some properties of the natural join operation? Is \bowtie associative? Is it commutative? Does it have an identity? Do we have another monoid here?

Chapter 11

Objects

11.1. Objects in programs

We have used the term “object” in previous chapters, but we have not yet defined exactly what we mean by “object”. The term has a specific, commonly-understood meaning in programming, although details of definitions and terminology vary from one writer to another and from one programming language to another. Here we will present informal definitions and terminology that are typical in the programming-language world, and that (in particular) agree with Python's treatment of objects.

An object is a piece of data that typically has a number of *attributes*, identified by name, much like fields in a tuple in a database relation (Section 10.5). For example, an object representing a person might have attributes “name”, “address”, and “department”. Objects in a program often represents things in the real world, and then an object's attributes are properties of the thing.

Thus a programming-language object is another kind of container for data, but one in which the elements of data are identified by name rather than by a position or a key. Often an object is mutable: values of its data attributes can be changed. For example, a program may allow the address attribute of a person object to be changed. The collection of the values of all the data attributes of an object at any time is called the *state* of the object. To say that an object is mutable, then, is to say that its state can change.

Other attributes of an object, besides the attributes that store data, are operations that can be performed on the object. These operations are called *methods*. A method of an object, like a data attribute of an object, “belongs to” the object. Methods of different objects, as well as data attributes of different objects, may have the same name but may mean different things and may be implemented differently. These facts are central to object-oriented programming, which we will describe in a bit more detail in Section 11.4.

In most programming languages an object is of a particular type or belongs to a *class* of similar objects. Object-oriented programming languages provide ways for programmers to define their own classes of objects. The type or class of an object determines what attributes (both data attributes and methods) the object has.

We can view an object as a mapping from attribute names to values, much as a relational-database tuple is a mapping from field names to values (recall Section 5.3). But, turning things around, we can view an attribute of a class as a mapping from objects of that class to values. For example, the attribute “name” might map a “person” object to a string, the name of that person. So, if our program needs a mapping from “person” objects to names, we have seen two options so far (Section 9.4): we could define a function that takes a “person” object as a parameter and returns a name; or we could build a data structure (such as Python dictionary) that takes a “person” object as a key and produces a name. Now we have a third option: to store the name as an attribute of the “person” object.

An object is more than a collection of attributes, though. In a typical programming language, an object has an *identity* that belongs to that object alone. The identity of an object never changes, even if the object is mutable; on the other hand, two objects may be of the same type or class and have all the same attributes and state but be distinguishable by their identities. (Often a programming language interpreter or compiler will use an object's location in memory as the identity of the object.)

In Python (and in many other programming languages), the syntax for accessing an attribute of an object is of the form *object . attribute*. For example, if *boss* is a “person” object, *boss . name* would be the “name” attribute of that object.

In the case of a method, *object . method* is a function. In Python, the syntax for calling a method is *object . method(arguments)*. When Python executes the call, the object is treated as one argument to the function (we will see how this works in Section 11.2). We say that we *apply* the method to the object, passing the values in the parentheses as additional arguments.

Typically, a method uses or operates on data attributes of the object; for example, the call `boss.changeAddress(addr)` might set the `address` attribute of `boss` to `addr`, and the call `boss.addressLabel()` might return a string containing values of data attributes of `boss` formatted in a way appropriate for printing as an address label. Notice that in the latter case the method call passes no additional arguments, which is not too unusual.

11.2. Defining classes

One key characteristic of object-oriented programming languages is that the languages allow programmers to define their own classes of objects. We will show how this is done in Python.

A class definition is a compound statement, whose header looks like this:

```
class classname :
```

The body of the class definition contains function definitions, which define methods, and possibly other statements.

A class definition, when executed, implicitly defines a function for creating objects of the class; these objects are called *instances* of the class. The function is called the *constructor* for the class, and has the same name as the name of the class.¹

Different instances of a class have different identities. The binary operator `is` compares the identity of two objects; an expression of the form `a is b` has the value `True` if `a` and `b` are the same object and `False` otherwise. The operator `is not` has the opposite meaning.

A class definition may contain a definition of a method with the special name “`__init__`”. The `__init__` method is used to initialize newly-created objects. When a constructor is called, it first creates an object and then calls the

¹This description is a slight simplification of what really happens in Python, but it will be good enough for our purposes.

`__init__` method if the class has one. Programmers find this feature of Python so useful that most programmer-defined classes have `__init__` methods.

The constructor passes the newly-created object to the `__init__` method as an argument. If the constructor is called with arguments, it passes those arguments to `__init__` as additional arguments.

Here's a simple example of a class definition:

```
class Person:
    def __init__(self, name, dept, address):
        self.name = name
        self.department = dept
        self.address = address
```

This code defines a class whose constructor is called with three arguments: a name, an address, and a department (possibly all strings). The constructor calls the `__init__` method with four arguments: the newly-created object followed by the three arguments passed to the constructor. In this `__init__` method, the object is called `self`. The word `self` is not a keyword and has no predefined meaning in Python, but it is a common programming convention among Python programmers to use the name in this way. (By the way, another common convention is to capitalize the names of programmer-defined classes.)

This `__init__` method sets the values of data attributes of the new `Person` object from the arguments passed to the constructor; this is a common thing for `__init__` methods to do.

In this example two of the data attributes have names that are the same as names of parameters to `__init__`. The two uses of “address” (for example) refer to different things; there is no ambiguity because of the way that Python interprets names. A class defines its own *name space*, which is a mapping from names to things bound to those names.² The name space of a class contains the names of data attributes and methods of the class and anything else created within the body of the class definition. A function definition creates another name space, which contains the function's parameters and any names of

² Python uses its own dictionaries to construct the mappings for its name spaces.

variables and other things created within the function body. These name spaces are distinct from the “global” name space, which contains the names of things, such as variables and functions, that are defined outside class definitions and function definitions.

In the statement `self.address = address` (for example), the “.” operator finds the class of `self`, which is `Person`, and looks for “address” in the name space of that class. Since “address” on the right-hand side of the assignment statement occurs without the “.” operator, it is not a reference to an attribute of a class, so Python looks for it in the name space of `__init__`. The two occurrences of “address” refer to completely different things, as if the programmer had used two different names.

Now suppose we create a `Person` object, perhaps like this:

```
boss = Person("Malone", "IT", "127 Spring")
```

The constructor is called, `__init__` is called, and then (for example) the value of `boss.name` would be “Malone”.

To define an ordinary method of a class, we write another function definition within the body of the class definition, like this:

```
class Person:
    def __init__(self, name, dept, address):
        self.name = name
        self.department = dept
        self.address = address

    def addressLabel(self):
        return self.name + "\n" \
            + self.department + " Department\n" \
            + self.address
```

To use the `addressLabel` method, we would apply it to a `Person` object, perhaps like this:

```
toTheBoss = boss.addressLabel()
```

Then the object `boss` is bound to the parameter `self` in `addressLabel`; in this example there are no additional arguments. As in `__init__`, it is conventional but not required to use “`self`” as the name of the first parameter to ordinary methods of a class.

The same program might contain a definition of another class, say `Building`, that has a different collection of attributes; for example, a `Building` might have a `name` and an `address` but not a `department`. And `Building` might have methods that `Person` does not have, and vice versa. Also, `Building` might have an `addressLabel` method, but its definition might be different from that of the `addressLabel` method of the `Person` class. There would be no ambiguity, because each class has its own name space.

11.3. Inheritance and the hierarchy of classes

A key feature of object-oriented programming languages is *inheritance*, which allows a programmer to create a new class based on an existing class. Typically, the new class is the old class with features added. Again, we will use Python to illustrate the concept.

Suppose that we want to define a class called `Customer`. A `Customer` object is like a `Person` object, but it has an additional attribute: the company that the customer represents.

We could define another class using a definition much like the definition of the `Person` class, but in which the `__init__` method takes an additional argument and creates another attribute of the newly-created object, perhaps like this:

```
class Customer:
    def __init__(self, name, company, dept, addr):
        self.name = name
        self.company = company
        self.department = dept
        self.address = address
```

But using inheritance we can reuse the definition of `Person` in the definition of `Customer`, by referring to `Person` in the header of the new class definition:

```
class Customer(Person):
```

Then we say that the class `Customer` inherits from the class `Person`. We say that `Customer` is a *subclass* of the `Person` class, and that `Person` is the *superclass* of the `Customer` class.

In the `__init__` method of the class `Customer`, we call the `__init__` method of class `Person` to do whatever initialization needs to be done for any `Person` object, and then we do any initialization that is specific to `Customer` objects. To get the `__init__` method of the `Person` class rather than the `__init__` method currently being defined, we apply `__init__` to the name of the class, passing the newly-created object and any other arguments that the `__init__` method of that class expects.³ This is how the code might look:

```
class Customer(Person):
    def __init__(self, name, company, dept, addr):
        Person.__init__(self, name, dept, addr)
        self.company = company
```

Then a `Customer` object has a `company` data attribute, and also all the attributes of a `Person` object. Among these is the `addressLabel` method, which now works the same way for a `Customer` object as for any other `Person` object.

But suppose we want the `addressLabel` method to work differently for a `customer` object. Perhaps we want the customer's address label to include the name of the customer's company. We can make this happen by *overriding* the method in the definition of the `Customer` class, adding something like this after the definition of `__init__`:

```
def addressLabel(self):
    return self.name + "\n" \
           + self.company + ", " \
           + self.department + " Department\n" \
           + self.address
```

In Python, *every* piece of data is an object. All of the built-in types, such as `int` and `list` and `set`, are actually classes. The functions with those names,

³ When used in this way, `__init__` is an example of what Python calls a “class method”. We will not discuss class methods further here.

which we have called “type conversion” functions until now, are actually the constructors for those classes.

There is a class that is more general than any other, called `object`. Instances of `object` have no properties except for being objects. Python's built-in types inherit from `object`. So does any programmer-defined class that does not explicitly inherit from another class; `Person` is an example.

Any call to the constructor `object()`, which takes no arguments, produces an object that is guaranteed to be different from every other object in the Python world. Such an object has its uses. For example, recall the definition of the generator function `zip` that we gave in Section 7.5:

```
def zip(X,Y):
    it = iter(Y)
    for x in X:
        y = next(it, None)
        if y == None:
            return
        else:
            yield (x,y)
```

Here we used `None` as an end marker for `next` to return when the iterator `it` terminates. We remarked that we needed to assume that `None` could not be one of the values in the sequence `Y`. Now we can write a version of `zip` that requires no such assumption:

```
def zip(X,Y):
    it = iter(Y)
    endMarker = object()
    for x in X:
        y = next(it, endMarker)
        if y is endMarker:
            return
        else:
            yield (x,y)
```

Here the test “`y is endMarker`” is false for any `y` that can possibly be an element of `Y`, even another object created by another call to `object()`, so this version of `zip` is completely general and safe.

Programmers usually use inheritance so that a subclass is a more specific category of things than its superclass. For example, a `Person` is a specific kind of object, and a `Customer` is a specific kind of `Person`. The program containing those classes could define a class for another specific kind of `Person`, say `Employee`, and that class would inherit from `Person`. In turn, `Employee` could have subclasses `HourlyEmployee` and `SalariedEmployee`; perhaps the former would have an attribute `hourlyRate` and the second would have an attribute `salary`, each attribute having an appropriate meaning.

The class `Building` might have its own subclasses, such as `Office`, `Laboratory`, and `Shop`. Then the classes in the program would form a hierarchy:

```
object
  built-in types
  Person
    Customer
    Employee
      HourlyEmployee
      SalariedEmployee
  Building
    Office
    Laboratory
    Shop
  other programmer-defined classes
```

Python uses the hierarchy of classes to find attributes of objects, whether the attributes are data attributes or methods. For example, suppose that `emp` is an instance of `SalariedEmployee`, and suppose that the program contains a reference to `emp.attr`. Then Python would first look for `attr` in the name space of `SalariedEmployee`. If there is no such attribute there, Python would look in the name space of `Employee`, and so on up the hierarchy of superclasses, and use the first binding that it finds.

11.4. Object-oriented programming

Object-oriented programming is an important style of programming in the modern software world. The subject is rather complex, and a comprehensive treatment of it is beyond the scope of this book, but here we will introduce some of the basic concepts of object-oriented programming as they relate to the subject matter of this book.

In object-oriented programming, much or all of the central data in a program is held in instances of programmer-defined classes. Each class defines the implementation of a kind of object, usually in terms of lower-level variables and data structures, which are the data attributes of the class. The class definition also includes definitions of operations on objects of the class, in terms of code that operates on the data attributes; these operations are the methods of the class. One advantage of object-oriented programming is that class definitions conveniently group the data of an object together with the methods that operate on that data.

Another advantage is that class definitions can isolate implementation decisions from the rest of the program. For example, suppose that we are writing a program that requires several multisets. We can define a `Multiset` class, with methods for the multiset operations that we will need, such as `sum` and `count` (see Exercise 1). The data structure that stores the members of the multiset will be a data attribute of the class, and the methods will operate on this data attribute.

As we saw in Section 9.5, there are at least two reasonable Python implementations for a multiset: a list or a dictionary. If the rest of the program accesses the data attributes of the `Multiset` class directly, all the code that does so will be different depending on which implementation we choose. But if the rest of the program accesses multisets *only* by using the methods of the class, the only code that depends on our implementation decision will be in the bodies of the methods. We can even change our implementation decision later, perhaps changing from the dictionary representation to the list representation or vice versa, and the only code that we will need to change will be in the bodies of the methods.

Most object-oriented programming languages allow a programmer to say that an attribute is *private*, meaning accessible only within the class definition. The body of a method of the class can access a private attribute, but code outside the class definition is prevented from doing so, enforcing the isolation of the attribute from the rest of the program. In Python an attribute is made private by giving it a name beginning (but not ending) with two underbar characters.⁴

If all data attributes of a class are private, the only access to instances of the class from the rest of the program is through the methods. The programmer has complete freedom to change the set of data attributes and how they represent the state of an object. If the set of methods and their meanings is unchanged, nothing in the code that uses the class will need to be changed, no matter how radical the change in the data attributes may be; only the bodies of some or all of the methods will need to be changed.

The case study that follows will illustrate these ideas.

11.5. A case study: moving averages

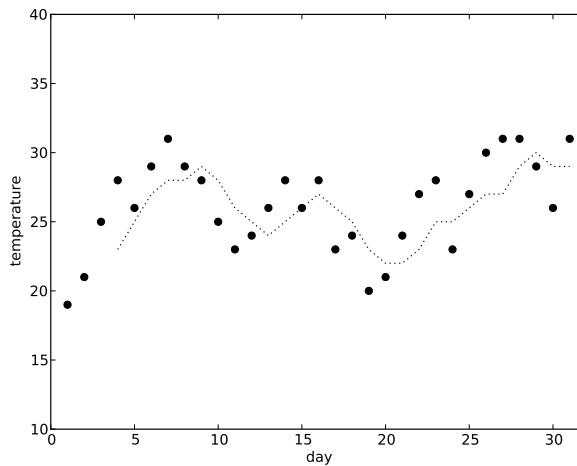
As a case study, let's develop a class definition to implement a moving average of a stream of data. The programming problem is very simple, but we will use the case study to illustrate the possible thought process of an experienced programmer developing a class definition using object-oriented programming.

A *moving average* of a stream of numbers (or at least the kind of moving average that we will discuss here) is the average of the most recent n numbers for some fixed n . Typically, the data in a moving average is a *time series* of data, which is a sequence of data values taken at equally-spaced instances in time. A moving average tends to smooth out the data and minimize the effects of random fluctuations and measurement errors. For example, in the U.S., economic statistics such as unemployment claims and housing starts are often reported weekly or monthly, as current numbers but also as moving averages over a period such as four weeks or three months. Some economists consider the moving averages to be the more meaningful numbers.

⁴ A devious programmer *can* access a private attribute in Python, but not in an obvious or convenient way.

The temperature data plotted with black circles in Figure 1.1 (page 15) is an example of a time series. Figure 11.1 shows the same data with a moving average shown as a dotted line; here n is 4. Notice that the moving average is not shown until there are at least n data points. Notice also that the moving average seems to lag behind the data points; this is because the moving average on each day is the average of the data for that day and the preceding $n-1$ days. (There is another kind of moving average that would use an equal number of data points before and after the day.)

Figure 11.1. Temperatures over a month, with moving average



Let us define a Python class of moving averages. We begin by deciding what operations we will need to perform on a moving average. We decide that we will need these methods:

- A constructor. Let us suppose that in our program we will need several moving averages, possibly over different numbers of data values. We decide to make the number of values a parameter of the constructor.
- A method to append a value to the sequence.
- A method to return the current value of the moving average.

Thus the class definition will look like this, in broad outline:

```
class MovingAverage:
    def __init__(self, nPoints):
        ...
    def append(self, newValue):
        ...
    def value(self):
        ...
```

Here is some simple code that uses the `MovingAverage` class. The generator function `smooth` iterates over the stream data and generates a second stream, the moving average. The number of points in the moving average is the second parameter of `smooth`.

```
def smooth(data, nPoints):
    avg = MovingAverage(nPoints)
    for value in data:
        avg.append(value)
    yield avg.value()
```

Before we fill in the bodies of the methods of the `MovingAverage` class, we need to decide on the representation that we will use for the data.

Conceptually, the data coming in is a sequence; we will give the data values to a `MovingAverage` object using a sequence of calls to the `append` method. As stored by the `MovingAverage` object, the values will apparently be a mutable sequence, so a Python list is the obvious choice to store them.

Should we store all the data values? If we do, the list will grow without limit. This may not cause performance problems in most situations, but it is possible that our code might be used in a program that is left to run for a very long time. It is good practice to avoid (whenever possible) memory usage that grows without limit, and we will never need to use more than the most recent n data values anyway. Let's plan to store only n values at most.

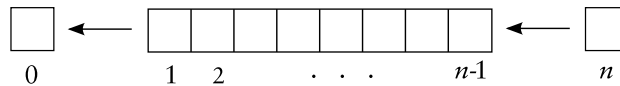
Here's a question that we will need to face before long: the moving average will be well-defined as soon as n values have been appended, but what happens

until then? We could define the moving average to be the average of the number of points that we have if that number is less than n , but what happens before any points at all have been appended? Let's assume, at least for now, that for our purposes it is acceptable to let the moving average be undefined until there are at least n data points.

Then let's use a data attribute to hold the sequence of up to n data points; the attribute will be a list, initially empty. Let's name the attribute “`__points`”; the two leading underbar characters make the attribute private to the class. We will also need to store the value of n ; let's store it as the private data attribute `__nPoints`. Then the `__init__` method looks something like this:

```
def __init__(self, nPoints):
    self.__points = [ ]
    self.__nPoints = nPoints
```

To implement the `append` method, we remove a value from the front of the list if it already contains n values, and then place the new value at the end of the list.



Here's the most obvious code to make both of those changes. Notice that each assignment statement creates a new list.

```
def append(self, newValue):
    if len(self.__points) == self.__nPoints:
        self.__points = self.__points[1:]
    self.__points = self.__points + [ newValue ]
```

But it is more efficient to modify the list in place if possible, and we can use the list `append` method instead of list concatenation to append the new value. (Notice that the `append` method that we are defining is distinct from the list `append` method; they are in different name spaces.) As it happens, Python has a method that will do the removing in place as well: it is `pop(i)`, where i is the position of the element to be removed. Using the `append` and `pop` list methods, the `append` of `MovingAverage` now looks like this:

```
def append(self, newValue):
    if len(self.__points) == self.__nPoints:
        self.__points.pop(0)
    self.__points.append(newValue)
```

And we can simplify this code a bit by creating an alias for `self.__points`. Recall (see Section 6.3) that the alias `points` refers to the same list object as `self.__points` does, so that modifying `points` modifies `self.__points` as well.

```
def append(self, newValue):
    points = self.__points
    if len(points) > self.__nPoints:
        points.pop(0)
    points.append(newValue)
```

The first draft of the `value` method might look like this. We return `None` if the moving average is undefined; it is the responsibility of the code that uses the `MovingAverage` class to test for `None` where necessary and do something appropriate.

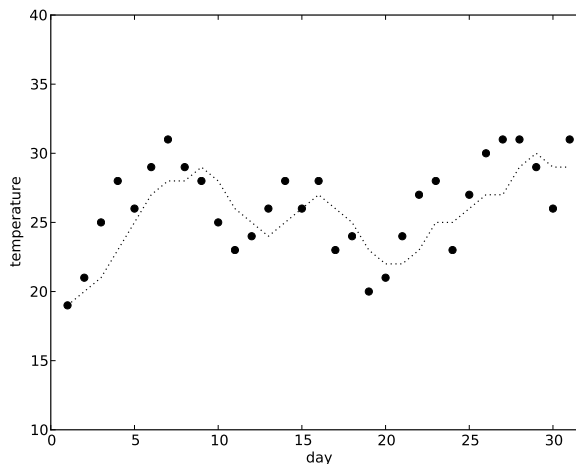
```
def value(self):
    points = self.__points
    length = len(points)
    if length == self.__nPoints:
        return sum(points) / length
    else:
        return None
```

Now notice one interesting fact: most of this code will work just as well if `length` is less than `self.__nPoints`, as long as `length` is not zero. In that situation we will be returning the average of fewer than n values, but for some purposes this value may be better than nothing.

```
def value(self):
    points = self.__points
    length = len(points)
    if length > 0:
        return sum(points) / length
    else:
        return None
```

Figure 11.2 is a version of Figure 11.1 with the moving average shown according to this extended definition.

Figure 11.2. Temperatures with an extended moving average



Let's make one more improvement to the code. Both `append` and `value` compute `len(points)` every time they are called. Most of the time this will be redundant computation; in fact, once the list contains n data values, the length will never change again. We can save the length as another hidden attribute, initialize the attribute appropriately and update it when the length of the list changes, and use the attribute most of the time instead of recomputing the length of the list. This tactic is similar to the technique of memoization that we saw in Section 9.4.

With these modifications, we have the class definition shown below as Example 11.1. Notice the documentation in comments; this is the sort of documentation that normally accompanies a class definition. The block of comments before the class definition describes the external interface of the class, the things that a programmer needs to know to use the class. The block of comments within the class definition describes internal details of the class, such as private attributes.

Example 11.1. The `MovingAverage` class

```
# MovingAverage: a moving average of a sequence
#                 of numbers.
#   MovingAverage(n) constructor, where n = number
#                   of values to average over
#   append(self, a)  append a to the sequence
#   value(self)     average of the n most recently
#                   appended values, or of all if
#                   fewer than n; None if none

class MovingAverage:

    # private attributes:
    #   __nPoints    number of values to average over
    #   __points     list of at most that many values
    #   __length     length of __points, kept current

    def __init__(self, nPoints):
        self.__points = [ ]
        self.__length = 0
        self.__nPoints = nPoints

    def append(self, newValue):
        points = self.__points
        if self.__length == self.__nPoints:
            points.pop(0)
        else:
            self.__length += 1
        points.append(newValue)

    def value(self):
        points = self.__points
        length = self.__length
        if length > 0:
            return sum(points) / length
        else:
            return None
```

So far so good. This implementation of `MovingAverage` will be perfectly good for many purposes. But will it be efficient enough for all purposes? We might be able to tell if we know how the class will be used.

Suppose that in a particular program the `value` method is called much more frequently than the `append` method. Perhaps the stream of data produces new values rather infrequently, as in the case of a weekly or monthly economic statistic. But perhaps the code is part of a popular web site that receives many requests per second for the value of the moving average.

In a situation like this, the `value` method often recalculates the average unnecessarily when the data values have not changed. We can eliminate this redundant computation by calculating the average whenever `append` is called and storing the value as another private attribute, which the `value` method can use when it is called. Again we are using memoization, and now `value` is implemented by accessing data instead of by performing a computation.

With this modification, `MovingAverage` might be defined as shown below in Example 11.2. The new data attribute is `__value`; notice that it doesn't even exist until `self.__length` is greater than zero, but we never use its value until then.

Version 2 of `MovingAverage` is not necessarily better than the original version in all situations. Suppose now that `append` is called much more frequently than `value`. Perhaps the stream of data is a physical measurement from a jittery sensor and arrives thousands of times a second, but the value of the moving average is required only once a second or once a minute. Then `append` computes the moving average many times when it is never used.

Thanks to object-oriented programming, though, the two versions of `MovingAverage` are completely interchangeable in any program. The external interface to the class, as documented in the blocks of comments before the class definition headers, is identical in the two versions. We can measure the efficiency of a program and substitute version 2 for the original version, or vice versa, to try to improve efficiency any time we like. No code that uses the class, such as the code in the `smooth` function (p. 187), would need to be changed in any way.

Example 11.2. The `MovingAverage` class, version 2

```
# MovingAverage: a moving average of a sequence
#               of numbers.
#   MovingAverage(n) constructor, where n = number
#                   of values to average over
#   append(self, a) append a to the sequence
#   value(self)    average of the n most
#                 recently appended values

class MovingAverage:

    # private attributes:
    #   __nPoints  number of values to average over
    #   __points   list of at most that many values
    #   __length   length of __points
    #   __value    average of values in __points
    # justification for memoizing __value: value() is
    # called much more frequently than append()

    def __init__(self, nPoints):
        self.__points = [ ]
        self.__length = 0
        self.__nPoints = nPoints

    def append(self, newValue):
        points = self.__points
        if self.__length == self.__nPoints:
            points.pop(0)
        else:
            self.__length += 1
        points.append(newValue)
        self.__value = sum(points) / self.__length

    def value(self):
        if self.__length > 0:
            return self.__value
        else:
            return None
```

11.6. Recursively-defined objects: trees

In the examples that we have seen so far, the values of the data attributes of programmer-defined classes have been values of built-in Python types, but they can also be objects that are instances of other programmer-defined classes. For example, consider the attribute `address` in the `Person` and `Customer` classes of Sections 11.2 and 11.3. The values of `address` were strings, but if we had an `Address` class (perhaps having its own attributes such as `street`, `city`, and so on), the values of `address` could be `Address` objects instead (with appropriate changes to the `addressLabel` methods, of course).

In fact, the value of an attribute can be an instance of the same class that is being defined. For example, suppose that we are defining a class `Person`. Two of the attributes of a `Person` object might be the person's mother and father. The values of these attributes might themselves be `Person` objects.

Then the `Person` class is defined partly in terms of itself, much as a recursive function is. We say that an instance of this class is a *recursively-defined object*. Recursively-defined objects are important in computer science, and knowing how to work with them is a useful programming skill.

Let's consider one kind of recursively-defined object: the tree. We saw trees briefly in Section 6.6. There are a number of ways of describing trees in mathematics and computer science, and we'll look at several of them, including recursive descriptions.

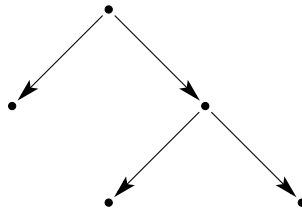
In the study of graphs, a tree is a kind of graph. A tree can be either a directed or an undirected graph. We will concentrate on directed trees of a particular kind: those that have a distinguished vertex called the *root* from which paths radiate. Trees of this kind can be defined by the following two properties:

- There is one and only one vertex with no edges leading to it. That vertex is the root of the tree.
- Each other vertex has one and only one edge leading to it.

These properties imply that for every vertex in the tree, there is a path from the root to that vertex, and no more than one. Paths in a tree never converge.

If a tree is finite (and we will consider only finite trees), it must have *leaves*, which are vertices with no edges leading from them.

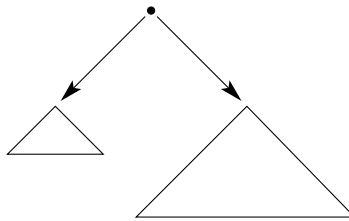
Here is an example of a tree. This one is unlabeled, and this time the root is at the top of the picture and the leaves are toward the bottom.



Now here is a recursive definition that defines trees of the same kind:

A tree is either

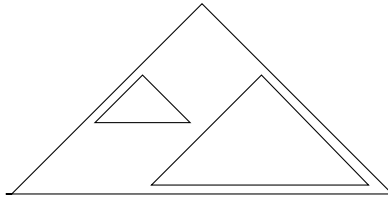
- a leaf, or
- a vertex (the root) with edges to one or more *subtrees*, which are trees defined in the same way.



Let's look at variations on the recursive definition of a tree. Here is one:

A tree is either a leaf, or an object containing one or more subtrees, which are trees defined in the same way.

Here, instead of speaking of edges to the subtrees, we speak of the subtrees as being contained in the larger tree. The trees are nested, much as expressions and compound statements in Python are nested.



This is the view that we often take in programming, especially when programming with objects. Instead of a tree being a graph with edges, it is an object that has other trees as attributes. The edges may actually be present anyway, as bindings or pointers, depending on how the programmer or the programming language implements the tree objects. Sometimes we think of the edges explicitly when visualizing a tree or drawing a picture of it, but often we don't.

Here is another definition that is slightly different:

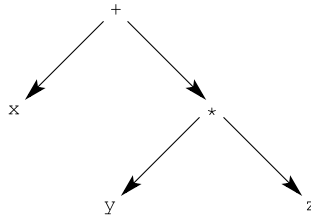
A tree is either empty, or is an object containing one or more subtrees, which are trees defined in the same way.

With this definition, a leaf is a tree all of whose subtrees are empty.

This definition allows a tree to be nothing at all, which makes no sense in many contexts. However, empty trees can be useful in programming, as we will see.

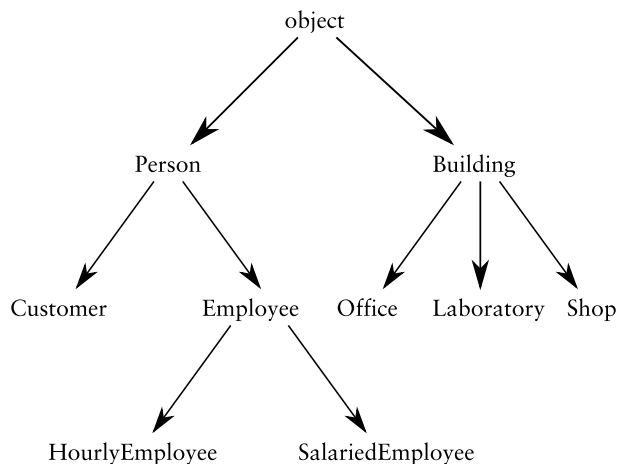
In computer science work, vertices of a tree are often labeled; in fact, sometimes a vertex has more than one piece of data attached to it. If trees are programmed as objects, each vertex is represented by a tree object, and its attributes are the labels and other data as well as the subtrees.

The data attached to different vertices can be of different kinds. Here is a typical example of a computer-science tree: it represents an expression.



The expression is “ $x + y * z$ ”; perhaps it is part of a program. The leaves are labeled with the names of variables, and the other vertices are labeled with operators. Programming-language interpreters and compilers often use similar trees to represent expressions and other parts of programs.

Trees are ideal for representing nested structures like expressions. They are also useful for representing information of many other kinds, including hierarchy relationships. The tree below represents a hierarchy of classes and subclasses like the one presented in Section 11.3. (This is an example of a tree that is not binary.)

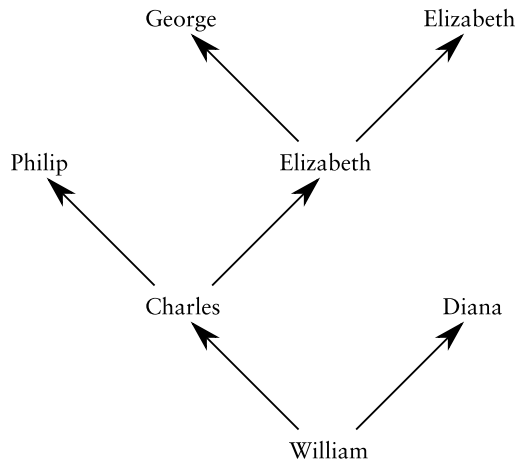


In common English usage there are “trees” that are actually trees in the mathematical sense, structures like those that we have been discussing. They are family trees.

There are two kinds of tree that are commonly called “family trees”: one shows the ancestors of a given person, and the other shows the descendants of a given person. We will consider the first kind, which we will call an “ancestry tree”.

Figure 11.3 shows an ancestry tree for a certain person named William. In this diagram older people are toward the top and younger people are toward the bottom, so the root of the tree is at the bottom. The edges point from a person to the person's father (up and toward the left) and mother (up and toward the right). Of course, the tree does not show all of William's ancestry; there are people in the tree (the leaves of the tree) whose fathers and mothers are not represented in the tree.

Figure 11.3. An ancestry tree



Notice the subtrees, which are trees of the same kind as the larger tree. For example, the tree whose root is labeled “Charles” is an ancestry tree for Charles.

Now let's see how we might represent such a structure as a recursively-defined object in a program.

Using Python as always, let's define a class called `Person`. Each instance of the class will represent a person in an ancestry tree, and will have the attributes `name`, `father`, and `mother`. Here are the class heading and the constructor:

```
class Person:
    def __init__(self, name, father, mother):
        self.name = name
        self.father = father
        self.mother = mother
```

Then we can construct William's ancestry tree by using code like the following. We construct `Person` objects for parents first, so that we can use those objects as values for the values of the `father` and `mother` attributes for younger people. We use the value `None` for the `father` and `mother` of people at the leaves of the tree; of course this means that the tree contains no information about the parents of these people, not that they have no parents!

```
george = Person("George", None, None)
elizabeth = Person("Elizabeth", None, None)
elizabeth2 = Person("Elizabeth", george, elizabeth)
philip = Person("Philip", None, None)
charles = Person("Charles", philip, elizabeth2)
diana = Person("Diana", None, None)
william = Person("William", charles, diana)
```

This code is only for illustration. In practice, especially for a larger tree, we would probably want to get the data from a file rather than hard-coding it into the program (see Exercise 4).

Now, once we have a tree or other recursively-defined object, how can we process it? Here is a very important principle:

A computation that processes a recursively-defined data object is likely to be recursive itself, with its structure following the recursive definition of the object.

For example, here, in broad outline, is the structure of many computations that process trees:

To *process* a tree:
if it is a leaf:
 do something with the data in the leaf
else:
 do something, possibly different, with the data in the root
 and recursively *process* each of the subtrees
 and combine the results

Not all computations on trees can be made to fit this pattern. But if a computation can, it is likely to be simpler and more obviously correct than a computation that does the same job in some other way.

Here's an example. Let's write a method for the `Person` class to construct a set of the names in the ancestry tree of a person.

We'll follow the above recursive pattern. Whether or not the `Person` is a leaf, we'll start by taking the name of the person, as a one-element set. Then, if the person's father is represented in the tree, we add the names in the ancestry tree of the father, and similarly for the person's mother. Notice that the “+” in the augmented assignment operator “+=” is set union.

```
def ancestorNames(self):  
    result = { self.name }  
    if self.father != None:  
        result += ancestorNames(self.father)  
    if self.mother != None:  
        result += ancestorNames(self.mother)  
    return result
```

This solution is not bad, but we can simplify it. Consider the recursive definition of a tree that allows a tree to be empty. It makes no sense for an ancestry tree for a person to be empty — it always includes the person at least — but suppose we let the `father` or `mother` attribute of a `Person` object be an empty tree? Then a value of `None`, instead of being a marker that indicates missing information, represents an empty subtree.

Now, instead of having to test each subtree for the `None` value, we simply let the empty tree be the basis case for the recursion. The set of names in an empty

ancestry tree is the empty set, of course. Following this line of reasoning, we arrive at the following solution:

```
def ancestorNames(self):
    if self == None:
        return set()          # the empty set
    else:
        return { self.name } \
            + ancestorNames(self.father) \
            + ancestorNames(self.mother)
```

This code is not only simpler, but more closely follows the pattern of code for a recursive function definition that we have often seen, going back to the factorial function of Section 4.2. The basis case and the recursive case are clearly separated, so that it is easier to see at a glance that the method body computes what we want.

This example illustrates two principles:

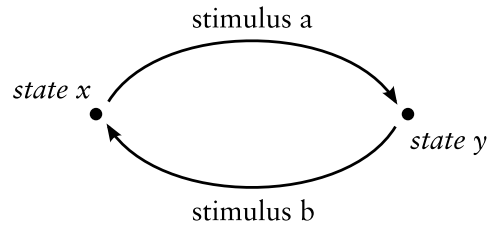
1. It's fine to choose a mathematical definition that makes your programming job easier, or to tweak a definition to your advantage.
2. Don't tweak the code and ignore the mathematical definition! Pick a definition, make sure that it says what you want, and then write the code to match the definition.

11.7. State machines

The state of an object is information that the object retains from one method call to another. A method may cause the object to change state (as the `append` method of a `MovingAverage` object does), and the result of a method call may depend on the object's state (as in the case of the `value` method).

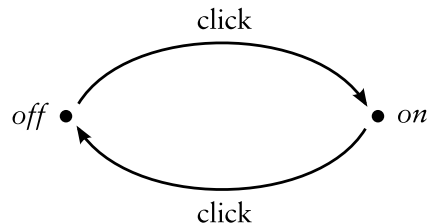
In designing objects in programs, it is sometimes useful to think explicitly of states and transitions between states. This is especially true when the number of possible states of an object is finite and rather small. Then we can give each state a name.

Furthermore, we can show the behavior of the object by drawing a *transition diagram*. This is a particular kind of a labeled directed graph, in which each vertex is labeled with the name of a state and each edge is labeled with the name of a stimulus or input that causes a transition from one state (at the tail of the edge) to another (at the head).



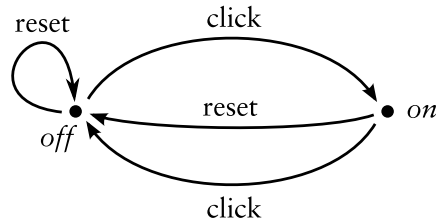
A transition diagram defines a *state machine*, which in computer science theory is a kind of mathematical object. Most mathematical properties of state machines are beyond the scope of this book, but we'll simply use state machines and their transition diagrams to help us describe and design objects. Software designers often do this, and so do other professionals such as hardware designers.

For example, consider the kind of electrical pushbutton that you click once to turn on and click again to turn off. We can describe it as a state machine having two states, *off* and *on*, and a single stimulus called "click" that causes a transition from each state to the other. Here is the transition diagram for that state machine:



Now consider a slightly more complicated kind of electrical device. When you click it, it works like the previous kind of pushbutton. But there's no way of

looking at the button to tell whether it is on or off, so the manufacturer has added a “reset” capability: when you press and hold the button down for several seconds, it turns off, whether it was on or off before. Here's the transition diagram for the corresponding state machine:



We can define a Python class whose objects behave like this device does, and we can use the transition diagram as a guide as we write the code. The state of the object will be stored in a private data attribute. There are only two states, so the attribute needs to have only two possible values. Boolean values are one obvious choice: we name the attribute `__on`, and we use the value `True` to represent the state *on* and `False` to represent the state *off*.

The `__init__` method sets the `__on` attribute to an initial value; we arbitrarily choose `False`. The effect of the “click” stimulus is implemented by a method with that name. It sets the value of `__on` to `True` if it was `False` and `False` if it was `True`; we could use an if-statement to do this logic, but the Boolean `not` operator does the same thing more easily. The “reset” stimulus is handled by another method, which simply sets `__on` to `False`. The method `isOn` returns the value of the attribute as a Boolean value. All this code is very simple; it is shown as Example 11.3.

Now let's look at an example of a state machine object that does a typical kind of programming job: finding fields in a string. Perhaps the string is a line from a flat file, as in Section 8.4, but in the current example the fields are separated by sequences of one or more space characters, rather than by commas as in a CSV file; there may also be spaces at the beginning and end of the string. A field may contain any character that is not a space. We want to deliver the fields of a string as a sequence, perhaps as a stream.

Example 11.3. The `Pushbutton` class

```
class Pushbutton:
    def __init__(self):
        self.__on = False

    def click(self):
        self.__on = not self.__on

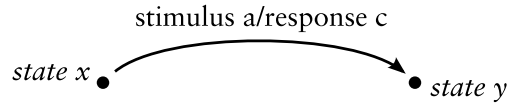
    def reset(self):
        self.__on = False

    def isOn(self):
        return self.__on
```

There are a number of ways of doing the job, but we'll show how to do it with the aid of a state machine. Here's the concept: the state machine will receive one character at a time from the string as a stimulus, and keep track of whether the current character is within a field or within a sequence of spaces; those will be the two states of the machine. The state matters, because a character can mean different things depending on its context; for example, a space coming after other spaces means nothing and can be ignored, but a space coming after non-space characters signals the end of a field.

The notation of transition diagrams has a number of variations, and we'll use two of them in this example:

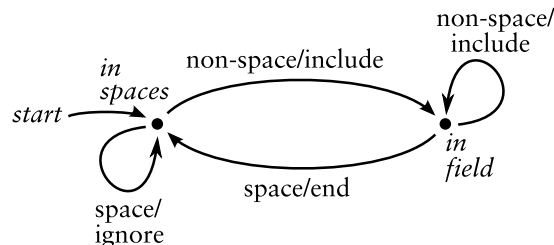
- An arrow pointing from outside the graph to one of the states, denoting the initial state of the machine; that is, the state that the machine starts in when it is created or turned on.
- An additional label on each transition, which is the name of a response to a stimulus. For example, a stimulus may be an input to a device, and the response would be the corresponding output. The response is what the machine does when it is in a particular state and receives a particular stimulus.



In our state machine, the stimuli will be characters from the string that we are taking apart, and the responses will be the “meaning” of each character in the context of the characters that have come before it. The different responses that we will need are:

- “include”: the character is part of a field.
- “ignore”: the character means nothing and can be ignored.
- “end”: the character signals the end of a field, but is not part of the field.

Here is the transition diagram for the state machine. The two states are labeled “*in spaces*” and “*in field*”. Since the string may start with spaces, the initial state is “*in spaces*”. In this state, a stimulus of a space character sends the machine back to the same state, and the response is “ignore”. Any other character is the first character of a field, so the machine goes to the state “*in field*” and the response is “include”. (You can think of the annotation “non-space” as indicating that this edge in the graph is a shorthand for many edges, one for each possible character that is not a space.) In the state “*in field*”, any character but a space sends the machine back to the same state, with the response “include”. A space signals the end of the field, so the machine goes back to the state “*in spaces*” and the response is “end”.



Now we can translate this transition diagram to Python code. There's one data representation decision to make: how will we represent the different

states of the machine and the different responses? Both collections are sets: they have no particular order and all elements are different. But it turns out that we won't need to do any set operations on these sets; we will only need to compare two values for equality. In a situation like this, we can simply represent every element of each set as a small integer value, as we noted in Section 8.5.

```
inField = 0
inSpaces = 1

include = 0
end = 1
ignore = 2
```

And now we write a class definition for the state machine that we want; the code is Example 11.4. The state of the machine is the data attribute `self.state`, and the constructor sets it to the initial state. We provide a method `advance` that takes a character as a stimulus and performs one state transition, setting the new state and the response depending on the old state and the character. We return the response as the value of the method call.

Example 11.4. A state machine for finding fields in a string

```
class FieldsStateMachine:
    def __init__(self):
        self.state = inSpaces

    def advance(self, char):
        if self.state == inField:
            if char == " ":
                self.state, response = inSpaces, end
            else:
                self.state, response = inField, include
        else: # self.state == inSpaces
            if char == " ":
                self.state, response = inSpaces, ignore
            else:
                self.state, response = inField, include
        return response
```

The code that uses the state machine is shown as Example 11.5. We define a generator function that yields fields of a string one at a time. We feed each character of the string to the state machine. We accumulate the characters of a field in the variable `field`, appending a character to it if the machine's response to the character is “include”. If the response is “end”, we yield the field and start a new one. When we reach the end of the string, the last field may not have been ended by a space at the end of the line; in this case, we yield that last field.

Example 11.5. Code that uses a `FieldsStateMachine`

```
machine = FieldsStateMachine()

def fields(string):
    field = ""

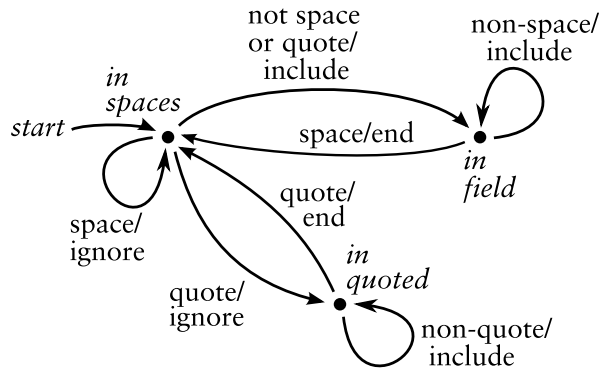
    for c in string:
        response = machine.advance(c)
        if response == include:
            field += c
        elif response == end:
            yield field
            field = ""
        # else response == ignore: pass

    if field != "":
        yield field
```

So far so good. But this example is very simple, and the code doesn't do anything we couldn't do more easily in other ways, such as with the Python `split` method. Now let's add a complication or, as we say in the software world, a “feature”.

Suppose that a field in the input file is allowed to contain spaces. Perhaps a field is the name of a city, so that cases like “Los Angeles” and “East St. Louis” must be allowed. Then when a field may contain a space, it is enclosed in double-quote characters, like this: “East St. Louis”. The double-quote characters are not considered part of the data that the field represents.

Now our state machine must contain a new state and some new transitions. In the state *in spaces*, a double-quote character causes a transition to the new state *in quoted*, meaning inside a quoted field; the double-quote character is ignored. The machine stays in that state until a double-quote character is received, which ends the field. The new state transition diagram is shown below; “quote” means a double-quote character.



The revised code for the state machine is shown as Example 11.6. The changes are easy to make, once we have the new transition diagram: we just need to add code for the new state and the new transitions. The body of `advance` may look long and complex, but its structure is quite regular, and we can easily compare the various parts to the corresponding parts of the transition diagram to make sure that we have written the code correctly.

By the way, computer scientists have observed that a monoid can be defined from transitions in a state machine, and computer science theory develops the implications of this fact. Those results are not immediately relevant in day-to-day programming, though, so we won't present the details here. Let's just use the observation to reinforce the idea that, in computing, monoids are everywhere.

Example 11.6. A state machine for finding fields, version 2

```
inField = 0
inSpaces = 1
inQuoted = 2

include = 0
end = 1
ignore = 2

class FieldsStateMachine:

    def __init__(self):
        self.state = inSpaces

    def advance(self, char):
        if self.state == inField:
            if char == " ":
                self.state, response = inSpaces, end
            else:
                self.state, response = inField, include
        elif self.state == inQuoted:
            if char == "'":
                self.state, response = inSpaces, end
            else:
                self.state, response = inQuoted, include
        else: # self.state == inSpaces
            if char == " ":
                self.state, response = inSpaces, ignore
            elif char == "'":
                self.state, response = inQuoted, ignore
            else:
                self.state, response = inField, include
        return response
```

Terms introduced in this chapter

attributes of an object
state of an object
class
identity of an object
instance

superclass
overriding a method
private attribute
recursively-defined object
root

constructor	leaf
name space	subtree
inheritance	state machine
subclass	transition diagram

Exercises

1. Write a Python class `Multiset` that is based on dictionaries. The class must implement the methods `isEmpty`, `count`, `intersection`, `union`, and `sum`.

In Python a programmer-defined class can inherit from a built-in class, so you might try making your `Multiset` class inherit from the `dict` class. What are the advantages and disadvantages of doing so?

2. Write a Python class `Monoid`. Give it at least the following methods:
 - An `__init__` method that takes two arguments, a two-argument function that implements the monoid operation and a value for the monoid identity. For example, here's how the class constructor might be called:

```
integerAddition = Monoid(lambda x,y: x+y, 0)
```

- A method `apply` that takes two parameters, applies the monoid operation (the function that was the first argument to the constructor) to them, and returns the result.
 - A method `reduce` that takes an iterable of values as a parameter and reduces its elements to a single value, using the monoid operation and the monoid identity, much as the `reduce` method of Section 6.4 does.
3. Write a variant of the `MovingAverage` class that is suitable for data that is not equally-spaced in time. The argument to the constructor, instead of being a number of data points, will be a duration (in appropriate units, such as seconds or days) over which the moving average is to be computed.

The argument to the method `append` will be an ordered pair of a time (in the same units) and a data value that corresponds to that time.

4. Design the structure of a file that a program could read to build an ancestry tree of the kind we saw in Section 11.6. Be aware that in an ancestry tree it is not too uncommon for two or more people to have the same name; a name does not uniquely identify a person in the tree.

Write a method for the `Person` class to read a file with this structure.

5.
 - a. Write a method `ancestors` for the `Person` class that constructs the set of all ancestors of a person (including the person), not just the set of ancestors' names. Perhaps there are several people with the same name but differing in other attributes that we haven't mentioned so far, such as birth and death dates; you want to include everyone who is an ancestor.
 - b. Now how can you find the set of ancestors of a given person, not counting that person? No, don't try to write a method that makes the original person a special case! Do it an easier way.
 - c. Using your `ancestors` method, how can you construct the set of ancestors' names? Do it with a single Python expression.
6. Add attributes `born` and `died` to the `Person` objects of Section 11.6; their values will be integers, representing the years in which the person was born and died. Use `None` for the value of `died` for a person who is still alive.

Write a method that takes `self` and a year as arguments, and returns the set of ancestors of a person (including the person) who were alive during the given year.

7. Modify the `Person` class of Section 11.6 so that it can be used to construct a tree of descendants of a person. Then a `Person` object will have attributes that are the children of the person rather than the parents. How will you represent the collection of children of a person?

Write a method for this class that constructs the set of descendants of a person, including the person.

8. Modify the `fieldsStateMachine` class of Section 11.7 to add one more feature: a quoted field may now contain a double-quote character as part of the data that the field represents. The two-character sequence “\” now represents a double-quote character within the field. Also, so that a quoted field can still contain a “\” character as part of the data that the field represents, the two-character sequence “\\” now represents a single “\” character within the field. In fact, to simplify things, let's say that a “\” character followed by *any* character represents the second character. Draw the new state transition diagram, and then write the corresponding code.
9. Design the structure of a file to encode the kinds of transition diagrams shown in Section 11.7: the states, initial state, stimuli, responses, and transitions. Now write a program that will read such a file and produce as output an advance method like the one in Example 11.6.

Such “code-generating” programs are used by writers of programming-language interpreters and compilers to help produce code for such jobs as finding the names, keywords, constants, operators and punctuation marks in lines of programs.

Chapter 12

Larger programs

12.1. Sharing tune lists

In this chapter we'll examine how we might apply the ideas presented in this book to larger programming problems. We won't present whole programs — they would take too much space in the book and would be tedious to read in their entirety anyway — but we'll sketch how the programs might be designed and, especially, how mathematical structures might be used in the designs.

Here's our first case study. The task is to build a program to make it easy for friends to share lists of their favorite tunes from those that they play on their computers and mobile devices.

The program will allow people to send recommendations to their friends, as you would expect, but it will also keep track of the songs that a person plays. It will send lists of the tunes that the person has played most often, perhaps once a week, to all of the person's friends through the wireless connections on their devices. It will also receive tune lists from the friends and compile them into a single list arranged by popularity. People can use these tune lists as recommendations, and look up the tunes in their own tune libraries, or (for example) look for other tunes by the same artists.

That's the statement of the problem in brief. Now let's see how we might design the program. We'll show some fragments of the actual Python code too.

A “tune list” is a sequence, in order by some measure of popularity, but not necessarily a list in the Python sense. Each tune has a number of attributes: artist, title, composer, perhaps genre, and so on. Our program will process the attributes of tunes, not the sound files themselves. A tune is identified by its collection of attributes. If all the tunes came from one source, such as a particular company's online music store, we could use something like the

catalog number in the store to identify a tune, but we assume that the tunes can come from a number of different sources, including home recording.

We'll represent a tune by a Python tuple, in which the first element is the artist, the second element is the title, and so on for as many attributes as we want to include. Other representations are possible (see the exercises at the end of the chapter), but the tuple representation will lead to a simple design. A tuple is an immutable object, so we can form sets of them and use them as dictionary keys. Let's call each of these tuples a “tune-tuple”.

Each attribute will be at a fixed position in every tune-tuple. To access attributes by name, we define a set of functions that map a tune to one of its attributes:

```
def artist(tune):
    return tune[0]

def title(tune):
    return tune[1]
```

and so on for all the attributes that we want to access by name.

A person's tune library will be represented by a set of tune-tuples, forming a relation if we want to think of it that way. The data will come from a file that is in a variant of CSV format: each line represents a tune and contains its attributes separated by vertical-bar characters (“|”). This format allows attributes such as tune titles to contain commas, which is not too uncommon. We can build the set of tune-tuples using a variant of the `setOfTuples` function from Section 8.4. Here is the definition we use; we simply split on vertical-bar characters instead of commas. Notice the set comprehension.

```
def setOfTuples(fileName):
    file = open(fileName)
    return { tuple(line.strip().split("|"))
            for line in file }
```

Then if the data file is named “myTunes”, we do this to build the set-of-tuples structure:

```
myTunes = setOfTuples("myTunes")
```

As a person plays tunes, the tunes are communicated to our program (by means that are beyond the scope of this book) as a stream of tune-tuples. Our program accumulates the tune-tuples in a multiset. We use a multiset, not a set, because of course the person can play the same tune more than once, and one of our goals is to count how often each tune is played.

We define a class of multisets, since (as you'll see) we'll need several of them. We implement a multiset as a mapping from values to counts, using a dictionary as we did in Section 9.5; we keep the dictionary in a private data attribute. We provide a `tally` method like the function of that name that we defined in Section 9.5.

```
class Multiset:
    def __init__(self):
        self.__count = {}

    def tally(self, value):
        if value in self.__count:
            self.__count[value] += 1
        else:
            self.__count[value] = 1
```

We create a `Multiset` object named `plays`, and as a person plays tunes, the tune-tuples are entered into it:

```
plays.tally(tune)
```

In the class `Multiset`, we provide another method to extract a tune list from the `plays` multiset. The method is called `topN`, and returns the n tunes with the most plays, highest first. Here n is an argument, to let the user set the number as a preference; there is nothing magical about the number ten. In any case, to get a play list of the top 10 tunes, we would write something like this:

```
playList = plays.topN(10)
```

There are several ways that `topN` could produce the required list. Here are two of them:

1. Produce the contents of `__count` as a sequence of ordered pairs in which the count comes first and the tune tuple comes second:

```
byCount = ( (count, tune)
            for (tune, count) in items(self.__count) )
```

Sort that sequence using the builtin function `sorted`, which produces a list in ascending order of count (and, within subsequences of equal count, in ascending order of tune-tuple, but that's not so important). Reverse the result (the Python library has a list method `reverse`), take a slice of the first n elements of the resulting sequence, and produce a sequence of just the tune-tuples from the ordered pairs using a comprehension.

2. The builtin function `sorted` can, with extra arguments, do most of the work and produce a list sorted by count in reverse order in one step. This method requires some Python that we haven't covered in this book, but if you're interested you can look in the Python documentation for the function `sorted` and the phrase “keyword argument”.

However `topN` does it, we get a sequence of the tunes that the user has played most often, and we transmit the sequence to each of the user's friends in the same CSV form that we use for the user's tune library. (As with the stream of the user's plays, we won't discuss how the program sends or receives tune lists.) The collection of friends is a set: perhaps a set of online addresses, or a set of objects each having a person's online address as an attribute.

Meanwhile, the program receives (we won't discuss how) tune lists and recommendations from the friends. A recommendation is a tune-tuple and a message (probably a comment about the tune) in some suitable format; the program represents the recommendation as an ordered pair of a tune-tuple and a string. The program collects the tune lists in a set called `friendsLists` and the recommendations in another set called `friendsRecommendations`.

The program will be able to display all the tune lists and recommendations, but the program will also be able to tabulate both and produce aggregated tune lists. Like the user's plays, these are multisets at first and are then converted into sorted sequences. Unfortunately, we can't write multiset comprehensions in Python, but we can still construct the multisets easily using

for-statements and `tally`. For example, to produce a list of the tunes with the ten highest number of mentions in the tune lists of the friends, here's how the code might look:

```
allMentions = Multiset()
for fList in friendsLists:
    for tune in fList:
        allMentions.tally(tune)
topTenMentions = allMentions.topN(10)
```

Or to produce a list of the ten tunes that are first in the greatest number of tune lists of the friends:

```
allMostPopular = Multiset()
for fList in friendsLists:
    allMostPopular.tally(fList[0])
topTenMostPopular = allMostPopular.topN(10)
```

Or to produce a list of the ten tunes recommended by the most friends (recall that the first element of a recommendation pair is the tune-tuple):

```
allRecommended = Multiset()
for rec in recommendations:
    allRecommended.tally(rec[0])
topTenRecommended = allRecommended.topN(10)
```

We could also produce an aggregated tune list of the tunes played most often by the friends, weighed by the number of plays of each tune. For this computation we would need the counts, of course, so we would have each friend's machine transmit pairs of tunes and their counts, pruned to the top ten or some such number, rather than just a tune list; in other words, another multiset. Then we could produce an aggregated tune list by doing a big multiset sum or union of all the multisets coming from the friends. We might or might not find such a list useful, because it would be heavily weighted toward the tastes of the friends who spent most time listening to tunes, but it might be an option worth considering.

Then the user might want to play the tunes on any of these aggregated lists, or to search for tunes like them. Code to determine whether a given tune is in the user's tune library would simply contain the Boolean expression

```
t in myTunes
```

Perhaps the user would like to see tunes in the tune library that have the same artist as a given tune. We could get the set of such tunes with a comprehension:

```
{ t for t in myTunes  
  if t.artist() == tune.artist() }
```

Notice that this is a relational selection operation (Section 10.5). You can probably think of similar computations that the program might do.

Now let's review the mathematical structures that this case study has used: multisets, certainly, but also sets and set comprehensions; n -tuples; sequences of various kinds; mappings such as `artist`, `title`, and `topN`; and relations. In each case, notice how the mathematical structure was a natural choice for the job to be done.

12.2. Biological surveys

Our next case study is a program to record data from biological surveys.

For our purposes, a biological survey is a count of organisms of different kinds in a particular place or at a particular time or both. One good example is the Christmas Bird Count in North America. Other examples might be a count of grasses and wildflowers in a meadow, or a count of fish passing a particular dam over some period of time.

Usually a participant in a biological survey records different kinds of organisms observed and how many of each, and our program will accumulate these counts. Sometimes, as perhaps in the case of grass plants in a meadow, a participant may not count individuals, but our program will still be able to count the number of participants that observed a particular kind of plant. Or the meadow might be divided into grid squares, and the program would count the number of squares in which an organism was observed. In any case, the program will accumulate counts, like the tune-list program in Section 12.1 does. But this time we won't represent the data explicitly as a multiset — that is, as a set of key-and-count pairs — because another structure for the data is more important, as we will see.

In biology, a “kind” of organism is called a “taxon” (plural “taxa”). A taxon is identified by a scientific name, usually derived from Latin, and it often has a common name in English or whatever the local modern language may be. Taxa are arranged into levels called “ranks”, from more general to more specific. For example, for birds the major ranks are as follows (scientific names are shown in italics):

- Class. Example: *Aves* (birds).
- Order. Example: *Strigiformes* (owls).
- Family. Example: *Strigidae* (typical owls, not barn owls).
- Genus. Example: *Bubo* (horned owls).
- Species. Example: *Bubo virginianus* (Great Horned Owl).

Often, as in the case of birds, there are also intermediate ranks such as superorders and subfamilies. Our program will not have a particular set of ranks built in, but will use whatever ranks are in the data that we are given.

Each taxon except those at the lowest rank, then, comprises one or more taxa from the next rank down. The class *Aves*, for example, comprises the *Strigiformes*, the *Gaviiformes* (loons), the *Sphenisciformes* (penguins), and two dozen or so more orders.

Our program will need to represent this hierarchical structure of taxa, because a survey participant doesn't always identify a specimen down to the species rank (“It was a hawk, but I couldn't tell what kind”). In some surveys (of insects, for example), identification of many specimens down to the species level is probably too hard for many survey participants, and we would expect that most of the counts would be of taxa above the species rank. In any case, the program must be prepared to keep counts of taxa at any rank.

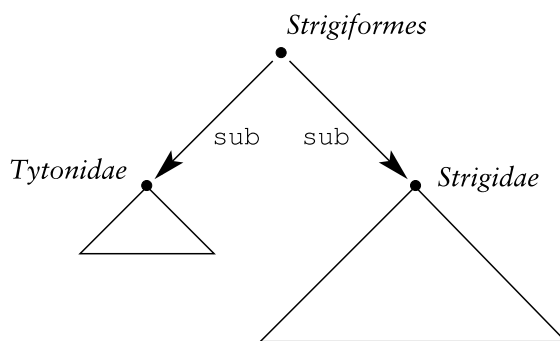
Now let's sketch the design of the program. The central structure is a `Taxon` class having the following attributes:

- A scientific name, `scientific`.

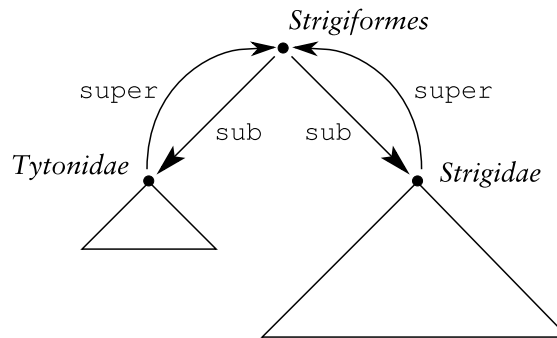
- A common name (optional), `common`.
- A rank, `rank`.
- A count, `count`.
- The taxon at the next rank up which contains the current taxon, `super`. It may be `None`, which happens when the program doesn't keep data for the taxon that would be at the next rank up. This would be the case for *Aves* in a survey of birds, for example. In any case, no taxon is contained in more than one higher-rank taxon.
- A set of subordinate taxa at the next rank down, `sub`. We assume that for our purposes the ordering is not important.

A `Taxon` with its `sub` set is a recursively-defined object, like the ancestry trees of Section 11.6, and so is the root of a tree. We don't require that all the `Taxon` objects be contained in one big tree with a single root; for example, in the survey of the meadow, there may be one tree containing all the grasses and another containing all the wildflowers. But each taxon is the root of some tree or subtree.

With the `super` attribute, the `Taxon` objects form a more complex recursively-defined structure. Recall from Section 11.6 that we can picture a tree as a directed graph; in a recursively-defined tree, an edge represents a binding or pointer from an object to another object which is an attribute of the first.



The `super` attribute adds edges in the opposite direction. Now we can think of the whole structure as a directed graph that is not a tree, but perhaps it is more useful to think of it as an undirected tree. The `sub` edges and `super` edges are paired, and each pair is like a single undirected edge that you can use to go in either direction.



We get the structure of the taxa from a data file, in CSV form as usual. Each line is a tuple defining a `Taxon` object and has the following fields:

- A scientific name, defining the `scientific` attribute.
- A common name, possibly empty, defining `common`.
- A rank, defining `rank`.
- The scientific name of the taxon at the next higher rank of which this taxon is a part, possibly empty, defining `super`.

The file line does not need to define the `sub` attribute, because it can be derived from the `super` attributes of the other `Taxon` objects.

For example, the line for *Bubo* might look like this:

```
Bubo,horned owl,genus,Strigidae
```

The taxa file is ordered by rank, highest first, so we can assume that the line for *Strigidae* (for example) comes before the line for *Bubo*. Other than this constraint, the lines of the file are unordered. But since the lines are not

completely unordered, the file does not quite represent a relation in the mathematical sense. (It happens to be an example of another kind of mathematical structure called a “partially ordered set”, which we haven't discussed in this book.)

The constructor of the `Taxon` class in the code (let's be careful not to confuse Python classes with biological classes!) is called with the fields of a file line as arguments, and the `__init__` method fills in the `scientific`, `common`, and `rank` attributes, and initializes `count` to zero. If the `super` parameter is not `Null`, `__init__` must also find the `Taxon` object with that name, set `self.super` to that object, and insert the current object into the `sub` set of that object. But how do we find the object?

Let's keep a dictionary `taxon` that maps scientific names to `Taxon` objects; the `__init__` method inserts an entry into it for the newly-created `Taxon` object. We keep a similar dictionary `taxonByCommon` that maps common names to `Taxon` objects; we'll see a use for it later.

Because of the ordering of the `taxa` file, when `__init__` is called the `Taxon` object for `super` already exists and `taxon` already contains the corresponding entry. (Throughout this case study, we won't show code that would deal with errors in the data, such as lines in the `taxa` file being in the wrong order or missing.) For the current `Taxon` object, we initialize the `sub` attribute to the empty set; it will be filled in as later lines of the `taxa` file are processed.

So here's what we have so far:

```

taxon = {}
taxonByCommon = {}

class Taxon:

    def __init__(self, scientific, common, rank, super):
        self.scientific = scientific
        self.common = common
        self.rank = rank
        self.count = 0
        self.sub = set()

        if super != "":
            superObject = taxon[super]
            self.super = superObject
            superObject.sub.add(self)
        else:
            self.super = None

        taxon[scientific] = self
        if common != "":
            taxonByCommon[common] = self

```

We give the `Taxon` class an obvious method to add a number to its count when a survey participant reports some data:

```

def add(self, n):
    self.count += n

```

So when a survey participant reports a count of some taxon, we use `taxon` or `taxonByCommon` (depending on which kind of name the participant has given us) to find the corresponding `Taxon` object, and apply `add` to that.

Now we come to the method that all this other structure and code is designed to support. The method computes the total count observed for some taxon, including the count stored for the taxon itself and the counts of all subordinate taxa all the way down the hierarchy. The computation, not surprisingly, is recursive. We add the `count` attribute of the taxon to the sum of the totals of all the elements of `sub`, and those totals are computed recursively in the same way. It's all very simple:

```

def total(self):
    return self.count + \
        sum((t.total() for t in self.sub))

```

Notice that this recursion is not explicitly defined by cases. The basis case of the recursion is reached at the lowest level of the taxon hierarchy, when `self.sub` is empty and so there are no recursive calls.

In the method, `sum` is a function that returns the sum of a sequence of numbers. Here is one possible definition of it, similar to the definition in Section 6.4 (where we also defined `reduce`):

```
def sum(seq):
    return reduce((lambda x,y: x+y), seq, 0)
```

Now we can produce a report of the counts of a given taxon and all its subordinate taxa, all the way down the hierarchy. For example, after a short time into a survey of birds, a report for the order *Strigiformes* might look like this:

```
8 Strigiformes (owl)
 7 Strigidae (typical owl)
  2 Asio (eared owl)
    1 Asio otus (Long-eared Owl)
    1 Asio flammeus (Short-eared Owl)
  0 Athene ( )
    0 Athene cucularia (Burrowing Owl)
  5 Bubo (horned owl)
    5 Bubo virginianus (Great Horned Owl)
 1 Tytonidae (barn owl)
  1 Tyto (barn owl)
    1 Tyto Alba (Common Barn Owl)
```

Here's a recursive `Taxon` method to produce such a report; the method produces a line for a taxon and then recursively produces a report for each subordinate taxon. Notice how the parameter `indent` is used to cause the lines for taxa at each lower level of rank to be indented two spaces further. Again, the recursion terminates when `self.sub` is empty.

```
def report(self, indent):
    print(indent + self.total() \
          + " " + self.scientific
          + " (" + self.common + ")")
    for sub in self.sub:
        sub.report(indent + "  ")
```

Now notice that once the program has read the whole taxa file, `values(taxon)` is the set of all the `Taxon` objects for all the taxa that the program has data about (see Section 9.2). It is actually a stream, but in no particular order, so we can think of it as a set. We can iterate over it, filter it, perform a computation on each element, materialize it as a set, and so on.

Here's one possible use for `values(taxon)`. In the taxa file for a survey, perhaps the data defines one big tree with a single root, as it might in a bird survey (the root would be the `Taxon` object for *Aves*). But perhaps it doesn't. In that case, it defines a set of trees (not surprisingly, the mathematical name for such a structure is “forest”).

We can get the set of roots of all these trees with a simple comprehension:

```
allTrees = { order in values(taxon)
              if order.super == None }
```

And then we can produce a comprehensive report of all the taxa counted, organized into the trees to which they belong:

```
for t in allTrees:
    t.report("")
```

As with the tune-list program in the previous section, you can probably think of other useful computations that we could add to the program.

Now suppose you were hired to write a program like this one. You complete the program using the design ideas above and submit it for approval. To your surprise, your boss asks for changes: there are a couple of additional requirements.

First, it seems that the taxa subordinate to a higher-rank taxon really do have an official ordering, called “taxonomic sequence”. The output produced by the `report` method, as the `Taxon` class is currently defined, would not be acceptable to a biologist; the lines need to be in taxonomic sequence.

Second, some taxa have more than one common name. For one thing, names of taxa change over time, and some survey participants might not know the most recent name for a taxon, or might be using old field guides containing

some old names. For another, a survey may want to treat two or more common names as equivalent for counting purposes, such as “Snow Goose” and “Blue Goose” (they are the same species, but the Blue Goose is a “color morph” of the Snow Goose).

The boss says, “Sorry that I didn't think of these details before, but you'll need to revise the program.”

Fortunately the changes are very simple. To meet the first new requirement, we use a list instead of a set for the container for subordinate taxa in a `Taxon` object. We change only two lines in the constructor. Here they are:

```
        self.sub = set()
and         superObject.sub.add(self)
```

We change those lines to these:

```
        self.sub = []
and         superObject.sub.append(self)
```

Then we simply require that lines in the taxa file be in taxonomic sequence. Then the `Taxon` objects are created in taxonomic sequence, and the objects are appended to the appropriate sub lists in taxonomic sequence.

Now, for example, the `report` method produces its output lines in taxonomic sequence as well. The `for`-statement in the method's body does not need to be changed at all, but now it iterates over a list rather than a set, and the list is in taxonomic sequence.

To meet the second new requirement, we use another data file that maps any additional common names to scientific names; it will be a CSV file as usual, and let's assume it is named `synonyms`. We will take each line of the file, convert it to a tuple using the `setOfTuples` function from Section 8.4, look up the scientific name in `taxon` to find the corresponding `Taxon` object, and add an entry to `taxonByCommon` for the new common name. The code is very simple and the iteration has a familiar look:


```
for (common, scientific) in streamOfTuples("synonyms"):  
    t = taxon[scientific]  
    taxonByCommon[common] = t
```

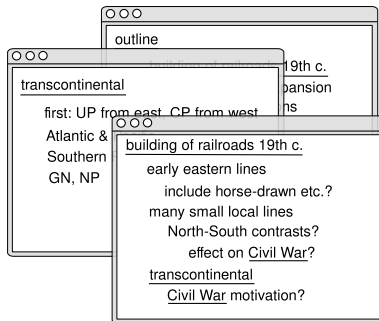
Only the official common name is stored in the `Taxon` object, but since all synonymous common names map to the same `Taxon` object in the `taxonByCommon` dictionary, they all contribute to the count of a single taxon. The boss is satisfied with this solution.

In this case study the central mathematical structure in the design is the structure of the taxa, which at its heart is a tree or (in the general case) a set of trees. The structure is made of recursively-defined objects. With the `super` attributes, the taxa structure is a directed graph, but we can also view it as an undirected tree. Mappings are also important in the design, especially `taxon` and `taxonByCommon`, the recursively-defined mapping `total`, and the file `synonyms`. The program also uses sets, as well as tuples and sequences of tuples.

12.3. Note cards for writers

Our last case study will be a program to help writers manage fragments of texts, such as outlines, notes, pieces of drafts and finished work, and citations. The program is designed for students writing papers and theses, and for academic and professional writers writing articles and perhaps even books.

The program will be based on the metaphor of a note card, of the kind that students and writers have traditionally used in their research. The program will show “cards” as windows on a computer screen:

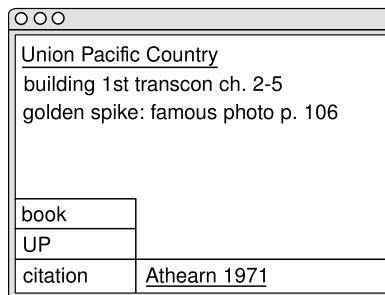


We join the software development project near its beginning. The organization developing the program has not completely decided on exactly what features the program will have or how the program will work, but has ideas. Here are some of them.

A user of the program will be able to create links among note cards, much like links among web pages. In fact, links will be able to point to several other kinds of things, such as:

- Web pages.
- Digital documents, in formats like PDF and Postscript, and specific places in documents.
- Entries in bibliographic databases.

A user will be able to annotate a note card with one or more keywords, and also with one or more key-value pairs. For example, in the illustration below, the note card has two keywords and one key-value pair. In the latter, the value is a link, perhaps to an entry in a bibliographic database.



The illustration shows a window titled "Union Pacific Country" with three control buttons in the top-left corner. The main text area contains the following text:

Union Pacific Country
building 1st transcon ch. 2-5
golden spike: famous photo p. 106

Below the text is a table with two columns and three rows:

book	
UP	
citation	Athearn 1971

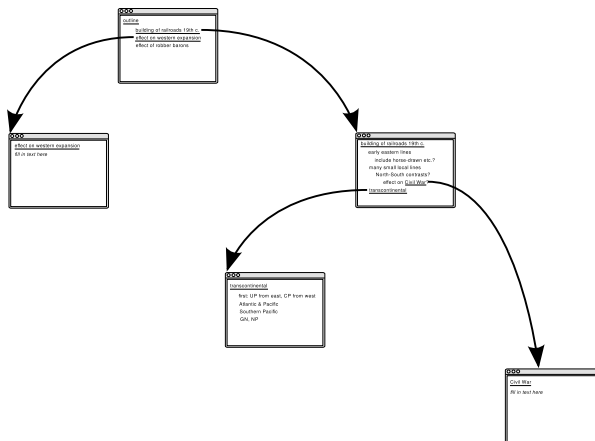
A user will be able to search for cards with particular keywords, or cards with particular values associated with particular keys. The program will also let the user perform other kinds of searches, such as for cards containing particular text.

Some keywords will be used to categorize cards as to their role, such as “outline” or “synopsis” or “draft”. Certain operations will most commonly

be done on cards having the same role; some such operations are splitting one card into two or more shorter cards, combining two or more cards into one longer card, and defining an ordering among cards.

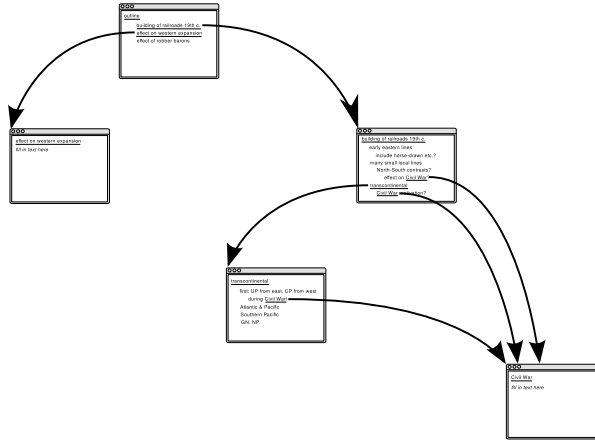
As a user works on a document, ordering fragments of text and combining them will become more and more important. The ultimate goal, after all, is to create a document as a single long sequence of text, probably from the contents of note cards having roles like “draft” or “final”. But as the work is in progress, the user might want to view the current state of all or part of the document by looking at a current outline or synopsis or draft taken from some or all of the note cards.

For example, consider the cards having the role “outline”. Suppose that one note card represents the top-level of an outline of a document, and suppose that the card contains links to other cards representing more detailed outlines of parts of the document. Those cards might contain links to other cards of the same kind, and so on. Then all these cards and links might form a tree:



A user might like to form one overall outline from this tree. Such an operation would produce an indented outline like the bird-survey report of Section 12.2, and could easily be programmed recursively, like the `report` method in that section.

But what if the links form a directed graph that is not a tree — that is, if there are note cards pointed to by more than one link?



What should the program do with cards like the one in the lower right? Should the program include its contents the first time it finds the card as it traverses the directed graph, or after it has processed all cards pointing to it? Or should the program include its contents more than once? And what should the program do if the user has somehow created a cycle in the directed graph? These are some of the design issues that the developers of the program are still debating.

In some situations the best solution may be to leave the ordering task to the user. For example, the program might display cards as if they were spread out on a table top, and let the user move them around using the program's graphical user interface.



This interaction may also be what the program does when the user performs a search: the program may take the set of cards satisfying the search criterion and display them as if on a table top, and let the user order them or perform other operations on them.

The developers of the program clearly have much more work to do and many more decisions to make, but let's take the description of the program as it stands and describe it in terms of discrete-mathematical structures.

We can describe a note card as a 4-tuple having the following components:

- A title, which is a sequence of characters.
- A body, which is text that may contain links.
- A set of keywords, possibly empty. Each keyword is a sequence of characters.
- A relation (the key-value pairs), also possibly empty. Each key is a sequence of characters, and each value is text that may contain links.

The designers of the program haven't yet decided how “text that may contain links” will be represented. One reasonable choice would be to represent a link as a particular kind of sequence of characters, much as in HTML. In HTML, a link to a location named “next” (say) is represented by a sequence of characters called a “tag”, having a form like “`Next`”. If some such representation is used in a note card for a link, then “text that may contain links” is just a sequence of characters much like any other.

Another reasonable choice might be to represent the links explicitly as different kinds of things, perhaps as Python objects. Then the representation of “text which may contain links” could be a sequence of elements from the union of two sets: the set of all strings and the set of all link objects. In either case, though, the representation is a sequence of some kind.

The collection of note cards is a set. The links among note cards imposes another structure on the set: a directed graph, in which the vertices are the note cards and the edges are the links. Some parts of the graph, such as the parts defining outlines, may be trees. Considering also the links to external

objects, such as web pages, all the links define a larger directed graph. The user may convert the contents of these graphs and trees, or parts of them, into sequences of text in various ways.

The result of a search is a set of note cards, a subset of the set of all note cards. The user may convert any of these subsets into a sequence.

To summarize, then: just in the description of the program that we have so far, we have tuples, sequences, sets, relations, trees, and directed graphs. Thus, like the other case studies, this one uses most of the kinds of mathematical structures that we have presented in this book. Certainly the programmers in the development organization will be able to use what they know about the mathematical structures to guide their thinking as development proceeds.

Exercises

1. Instead of representing a tune as a tuple in the program of Section 12.1, we might define a class of `Tune` objects and represent each tune as an instance of that class. What would be some advantages and disadvantages of doing so?
2. Or we might represent a tune as a dictionary mapping names of attributes to the corresponding attribute values. What would be some advantages and disadvantages of this representation?
3. Choose one of the programs presented in this chapter. Propose a new feature to add to the program, a feature that would use some kind of mathematical structure that the program doesn't already use. Be sure that the mathematical structure is appropriate for job it needs to do.

Afterword

I hope that you now appreciate some of the ways that the concepts and structures of discrete mathematics can be useful in programming.

We have only begun to explore the structures of discrete mathematics that find applications in programming. Among these are many specialized kinds of trees and graphs, special kinds of relations such as partial orders, and other variations on the structures that we have seen.

The Boolean values “true” and “false” are discrete-mathematical objects too. Logic is a central area in discrete mathematics, and of course computing depends on logic in any number of ways.

Sets of strings defined in various ways are called “formal languages” by computer scientists, who have studied them extensively. The technology in programming-language interpreters and compilers is based on the results, as well as on results from the study of state machines, trees, and graphs.

Another important area of discrete mathematics is called “combinatorics”; simple examples of results from combinatorics are numbers of combinations (Example 7.1) and heights of balanced trees (Section 6.6). Combinatorics is used in algorithm analysis, to predict running time of algorithms or to compare different algorithms for efficiency.

If you are a computer science student, you will see topics like these throughout your studies. You may take classes specifically called “discrete mathematics”, or classes with names like “combinatorics”, “formal languages and automata”, or “analysis of algorithms”. And your classes in such areas as operating systems, computer networks, data bases, and compiler writing will be full of discrete mathematics.

Whether or not you are a CS student or a computer scientist, you can use discrete-mathematical thinking routinely in your day-to-day programming work, and I encourage you to do so. Very often, your job will be easier and your solutions will be cleaner, and you may even enjoy your work more.

Solutions to selected exercises

Chapter 2, *An overview of Python*

2. On my computer, the Python interpreter runs for about two and a half minutes and then displays this, among other things:

```
MemoryError
```

This is an indication that the interpreter has used all the main memory available to it and still wasn't able to complete the computation.

4. For a positive integer n , $\log_{10}n$ is an approximation for the number of decimal digits it would take to write out n , just from the definition of a logarithm. It's a good approximation if n is large.

Now $\log_{10}n = (\log_2n)/(\log_210)$. If $n = 2^{2^{100}}$, $\log_2n = 2^{100}$, and $\log_210 = 3.32$ approximately.

So the number of digits it would take to write out $n = 2^{2^{100}}$ is $2^{100}/3.32$ approximately. By using the Python interpreter or some other calculator program on our computer, we find that 2^{100} is

1,267,650,600,228,229,401,496,703,205,376

So that number divided by 3.32 is approximately the number of digits you would need. Now do you believe that $2^{2^{100}}$ is a number too big for your computer to work with?

5. Yes; no. Try some examples. We'll look at such properties of sequences in more detail in Section 6.1.
6. Combining ideas from the first and second script in this chapter, we can get something like this:

```
file = open("names")
for line in file:
    firstname, lastname = line.split(" ")
    if firstname == "John":
        print(line)
```

You might not have thought of this solution if you didn't know (or guess) that " " can be used in Python to represent a blank space, but in fact it can. A space character — what you get when you press the space bar on your keyboard — is a character just as a letter or digit or punctuation mark is.

Chapter 3, *Python programs*

1. Here is one solution:

```
file = open("data")
for line in file:
    n = int(line)
    print "#" * n + " " + str(n)
```

2. Here is one solution:

```
file = open("data")
for line in file:
    n = int(line)
    if n > 20:
        barLength = 20
    else:
        barLength = n
    print "#" * barLength + " " + str(n)
```

3. Here is one solution:

```
file = open("data")
for line in file:
    n = int(line)
    if n > 20:
        print "#####/ ##### " + str(n)
    else:
        print "#" * barLength + " " + str(n)
```

Chapter 6, Sequences

3. No. Exponentiation is not associative: for example, $2^{(2^{**}3)} = 2^8 = 256$ but $(2^{**}2)^{**}3 = 4^3 = 64$. Furthermore, there is no identity for the `**` operator. 1 works as an identity on the right, since $n ** 1 = n$ for any value of n , but not on the left, since $1 ** n$ does not equal n for most values of n .

6. Here are two possible solutions:

```
def filter(test, sequence):
    result = ( )
    for a in sequence:
        if test(a):
            result += ( a, )
    return result
```

```
def filter(test, sequence):
    result = [ a for a in sequence if test(a) ]
    return tuple(result)
```

Which do you think is better? Why?

10. That version of `reduceRight` will pass f its arguments in reversed order. It gives the correct result only if f is commutative.

Chapter 7, Streams

2. Here is one solution:

```
def prefix(n, stream):
    i = 0
    for a in stream:
        if i >= n:
            return
        else:
            yield a
            i += 1
```

6. Not in the obvious way, because averaging is not associative. For example, suppose $\#$ is a binary averaging operator, so that $a \# b$ is defined to be equal to $(a + b)/2$; then $(a \# b) \# c \neq a \# (b \# c)$, as you can see by trying

some examples. In general, an average of averages is not equal to a single average of all the values.

As a further exercise, discuss how you might get around this problem in the case of the weather stations. Assume that you want to minimize the quantity of data that you transmit among computers, so that transmitting all the data to the central site and averaging the data there is not a solution.

Chapter 8, *Sets*

1. You want to apply a “big intersection” operation to the set of sets. Use the version of `reduce` that you wrote for Exercise 9 of Chapter 6. Did you write that function using a `for`-statement to iterate over the sequence? If so, you should be able to use the function on a set of operands just as well as on a sequence of operands — do you see why?

Chapter 9, *Mappings*

1. Pairs of values x and y of types for which the binary operator `*` is overloaded.
2. When the domains of c and d have no elements in common; that is, when $\text{dom } c \cap \text{dom } d = \emptyset$.

Chapter 10, *Relations*

2. We can define the set of vertices of the graph as the set of all a and b of all pairs (a, b) in the relation. However, we could add any other elements at all to the set; they will represent isolated vertices, but the graph will still have the given relation as its adjacency relation. The usual and sensible definition omits isolated vertices.
5. No, because the definition of a monoid would require that *any* two paths in the graph can be concatenated, whether they share an endpoint or not. But the set of all paths in a given directed graph, with concatenation, does form another kind of mathematical structure called a “category”.

6. We give only the solution for $\sigma_p(X)$ (the easy part). Let Y be $\sigma_p(X)$, and choose the same ordering of attributes in its representation as for X . Then $A_Y = A_X$, $col_Y = col_X$, and $R_Y = \{ x \in R_X \mid P(x) \}$.

Chapter 11, *Objects*

5. a. Just take one of the versions of `ancestorNames` in Section 11.6, rename the method to `ancestors`, and replace `self.name` with `self` everywhere.

- b. Use a set operation! If the person is `p`, the set you want is just

```
ancestors(p) - { p }
```

- c. Use a set comprehension, of course. It will take the form

```
{ ... for a in ancestors(p) }
```

Surely you can fill in the rest.

Index

A

abs, 38, 41
abstract algebra, 59, 61
add, 112
adjacency relation of a graph, 163, 165
alias, 65
antisymmetric relation, 155
append, 66
applying a method to an object, 176
argument, 23, 41
 passing, 23, 45
assignment, 18
 augmented, 21
 simultaneous, 53
 statement, 18
associative property, 14, 58, 59, 77, 104, 109, 130
attribute
 object, 175
 relational database, 167
augmented assignment, 21

B

basis case, 44
batch of updates, 130
“big” operator, 62, 69, 130
binary
 operator, 20, 49, 59
 relation, 55, 153, 154
 tree, 78, 197
binding, 18, 41, 45, 47, 47, 65
body, 30
Boolean, 32, 60

bowtie operator, 171

C

calling a function, 23
chutney, 129
class, 176
 constructor, 177
 instance, 177
closure
 reflexive, 167
 reflexive transitive, 167
 symmetric, 167
 transitive, 165, 167
Collatz algorithm, 91
comma-separated values, 55, 119, 120, 133
comment, 31, 190
commutative property, 14, 58, 104, 109, 130
complement, set, 109
complex plane, 51
composition
 of mappings, 131
 of relations, 156, 171
compound statement, 30
 body, 30
 header, 30
comprehension
 dictionary, 132
 list, 73-74, 74
 set, 111
 tuple, 74, 74
concatenation, 26, 58-59, 59, 62, 66
 of streams, 92-95, 95
conditional, 31-35, 31, 35
connected graph, 165
constant, 20
constructor, 177

container, 110
coordinates, 51, 51
count, 66, 147
count in a multiset, 146
CSV (see comma-separated values)
curried function, 129, 134
Curry, Haskell, 129
cycle in a graph, 165

D

database, relational (see relational database)
delimiter, 54
derivative, 46, 47
dictionary, 110
 comprehension, 132
 display, 131
difference (complement)
 relational database, 168
 set, 109, 129
directed graph, 160
discard, 112
display
 dictionary, 131
 list, 64
 set, 110
 tuple, 64
distributed processing, 103
distributive property, 61
domain of a mapping, 128, 131, 140
 restriction, 129

E

edge, 160
 multiple, 161
empty
 set, 108, 109, 111, 112

 string, 26
 tuple, 53
endless stream, 90-92, 92, 93, 102, 102
equivalence relation, 155
evaluating an expression, 20
exception handling, x, 5
expression, 20, 30
 generator, 74, 84, 85
 lambda, 48, 49

F

factorial, 43, 127
Fermat's Last Theorem, 110
Fibonacci numbers, 86, 143-145, 145
file, 5, 5, 11, 17, 54-56, 56
 flat, 54, 118-122, 122, 133
filter, 68, 76
finite but far too large, 22, 22, 22, 140
floating-point number, 21, 23-25, 25, 60
for-statement, 35, 67, 83, 110, 120
free monoid, 62
frozen set, 113
function, 13, 23, 30, 127
 argument, 23, 41
 as value, 45-48, 48
 call, 23, 41, 42
 curried, 129, 134
 definition in Python, 41-50, 50
 generator, 85, 90, 95
 higher-order, 48, 67, 72
 identity, 156
 parameter, 41
 recursive, 43-45, 45
functional programming, 72

G

generator

expression, 74, 84-85, 85

function, 85-90, 90, 95

graph, 159-167, 167

adjacency relation, 163, 165

connected, 165

cycle, 165

directed, 160

edge, 160

labeled, 163

loop, 161

multiple edges, 161

of a relation, 163

path, 164, 167

undirected, 160

vertex, 160

weighted, 164

group, ix, 61

H

hashing, 113, 132

header of a compound statement, 30

higher-order function, 48, 67-72, 72

I

identity, 58, 59, 61, 109

of an object, 176

identity relation, mapping, function, 156

if-statement, 31

infinite

number, 22

set, 109

value in Python, 22, 24, 60, 142

inheritance, 180

instance of a class, 177

integer, 8, 19-22, 22, 60

intersection

multiset, 146

relational database, 168

set, 109, 129

is operator, 177

iter, 100

iterable, 110, 111

iteration, 35-39, 39

iterator, 36, 83

J

join, natural, 170

K

key, 132

keyword, 30

L

labeled graph, 163

lambda expression, 48-49, 49

language, programming

functional, 72

object-oriented, 3

scripting, 3

very high-level, 3

leaves of a tree, 77, 195

len, 66, 83, 112

list, 64, 67, 111, 113, 118

comprehension, 73, 74

display, 64

loop in a graph, 161

M

map, 67, 76, 88

maplet, 127, 128, 132

mapping, 13, 127-152, 152
 composition, 131
 domain, 128, 131, 140
 domain restriction, 129
 identity, 156
 override, 129
 powers, 131
 range, 128

materialize, 102

max, 60

member of a set, 107, 113

memoization, 143, 157, 190, 192

method, 42, 175

min, 60

monoid, ix, 59-63, 63, 66, 69, 77, 95,
109, 130, 150, 173

 free, 62

moving average, 185

multiple edges in a graph, 161

multiset, 14, 145-149, 149

 brackets, 145

count, 146

 intersection, 146

 sum, 146

 union, 146

mutable, 65, 112, 113, 140, 175

N

n-ary relation, 153

n-tuple, 52, 57

name, 18

name space, 178

natural join, 170

nesting, 34, 196

Newton's Method, 38

next, 100

None, 101, 142, 189, 199

null string, 26

O

object, 19, 65, 175-212, 212

 attribute, 175

 mutable, 65, 112, 113

object-oriented

 language, 3

 programming, 183-193, 193

open, 4, 5, 36, 84

operand, 20

operator, 20

 “big”, 62, 69, 130

 binary, 20, 49, 59

 unary, 20

ordered pair, 12, 51, 127

overloading, 26, 42

override

 of a mapping, 129

 of a method, 181

P

pair, ordered, 12, 51, 127

parallel processing, 74-79, 79

parameter, function, 41

partial application, 47

partition, 155

passing an argument, 23, 45

path in a graph, 164-167, 167

pointer, 65

powers

 of a mapping, 131

 of a relation, 156

predicate, 154, 154, 157

print, 5, 9, 30

private attribute, 185

procedure, 41

processing
 distributed, 103
 parallel, 74, 79
 processor, 75
 projection, relational database, 169
 proper subset, 109

Q

quadruple, ordered, 51

R

range of a mapping, 128
 range, range object in Python, 36, 57, 67, 68, 83, 101, 103, 124
 readline, 38
 real number, 23, 25, 60
 recursion, 43-45, 45, 72
 recursively-defined object, 194
 reduce, 68, 77, 79
 reflexive
 closure of a relation, 167
 relation, 155, 158
 transitive closure of a relation, 167
 relation, 13, 153-173, 173
 antisymmetric, 155
 binary, 55, 153, 154
 composition, 156, 171
 equivalence, 155
 graph of, 163
 identity, 156
 n-ary, 153
 on a set, 154
 powers, 156
 reflexive, 155, 158
 reflexive closure, 167
 reflexive transitive closure, 167
 symmetric, 155, 157
 symmetric closure, 167
 transitive, 155, 159
 transitive closure, 165, 167
 relational database, 54-56, 56, 167-171
 attribute, 167
 difference, 168
 intersection, 168
 natural join, 170
 projection, 169
 selection, 169
 union, 168
 return, 24, 46
 return-statement, 41
 root of a tree, 77, 194

S

script, scripting language, 3
 selection, relational database, 169
 semigroup, ix, 61, 109
 sequence, 10, 57-81, 81, 110
 in Python, 64-67, 67
 set, 10, 107-125, 111, 125
 comprehension, 111
 difference (complement), 109, 129
 display, 110
 empty, 108, 109, 111, 112
 frozen, 113
 infinite, 109
 intersection, 109, 129
 union, 109, 129
 universal, 107, 109, 112
 side effect, 30
 simple statement, 30
 simultaneous assignment, 53
 slicing, 66, 66
 split, 7, 96, 119
 state, 175

state machine, 201-209, 202, 209

statement, 29-40, 40

body, 30

compound, 30

continuation with `\`, 29

for, 35, 67, 83, 110, 120

header, 30

if, 31

return, 41

simple, 30

while, 37

yield, 85, 89

stream, 83-106, 106, 110, 118

endless, 90, 92

string, 18, 25-26, 25, 26, 61, 66

subclass, 181

subroutine, 41

subscript, 54, 64, 66

subset, 108

proper, 109

substring, 32

subtree, 195

sum, multiset, 146

superclass, 181

symmetric

closure of a relation, 167

relation, 155, 157

T

time series, 185

transition diagram, 202

transitive

closure of a relation, 165-167, 167

relation, 155, 159

tree, 77, 194-201, 201

binary, 78, 197

triple, ordered, 51

tuple, 51-56, 52, 56, 66, 67, 67, 111, 119

comprehension, 74, 74

display, 64

two-dimensional array, 134

type, 18, 41

U

unary operator, 20

undirected graph, 160

Unicode, xii, 25, 145

union

multiset, 146

overriding, 129

relational database, 168

set, 109, 129

universal set, 107, 109, 112

universe of discourse, 107, 112

unpacking, 53, 64, 120, 121

V

value, 18, 45

variable, 19

vertex, 160

very high-level language, 3

W

weighted graph, 164

while-statement, 37

white space, 96, 96, 136

Wiles, Andrew, 110

Y

yield-statement, 85, 89

Z

zip, 99