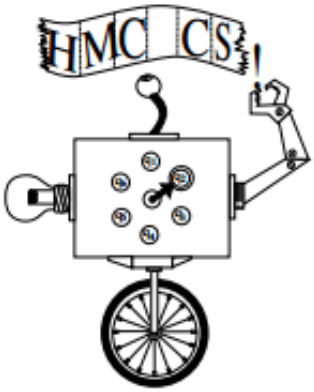
CS for All

Christine Alvarado (UC San Diego), Zachary Dodds (Harvey Mudd), Geoff Kuenning (Harvey Mudd), Ran Libeskind-Hadas (Harvey Mudd)



To The Reader

Welcome! This book (and course) takes a unique approach to "Intro CS." In a nutshell, our objective is to provide an introduction to *computer science* as an intellectually rich and vibrant field rather than focusing exclusively on *computer programming*. While programming is certainly an important and pervasive element of our approach, we emphasize concepts and problem-solving over syntax and programming language features.

Our name is Mudd!

This book is a companion to the course "CS for All" developed at Harvey Mudd College. At Mudd, this course is taken by almost every first-year student—irrespective of the student's ultimate major—as part of our core curriculum. Thus, it serves as a first computing course for future CS majors and a first and last computing course for many other students. The course also enrolls a significant number of students from the other Claremont Colleges, many of whom are not planning to major in the sciences or engineering. At other schools, versions of this course have also been taught to students with varying backgrounds and interests.

The emphasis on problem-solving and big ideas is evident beginning in the introductory chapter, where we describe a very simple programming language for controlling the virtual "Picobot" robot. The syntax takes ten minutes to master but the computational problems posed here are deep and intriguing.

The remainder of the book follows in the same spirit. We use the Python language due to the simplicity of its syntax and the rich set of tools and packages that allow a novice programmer to write useful programs. Our introduction to programming with Python in Chapter 2 uses only a limited subset of the language's syntax in the spirit of a functional programming language. In this approach, students master recursion early and find that they can write interesting programs with surprisingly little code. Chapter 3 takes another step in functional programming, introducing the concept of higher-order functions.

Chapter 4 addresses the question “How does my computer do all this?” We examine the inner-workings of a computer, from digital logic through the organization of a machine and programming the machine in its native machine and assembly language.

Now that the computer has been demystified and students have a physical representation of what happens “under the hood,” we move on in Chapter 5 to explore more complex ideas in computation and, concomitantly, concepts such as references and mutability, and constructs including loops, arrays, and dictionaries. We explain these concepts and constructs using the physical model of the computer introduced in the previous chapter. In our experience, students find these concepts are much easier to comprehend when there is an underlying physical model.

Chapter 6 explores some of the key ideas in object-oriented programming and design. The objective here is *not* to train industrial-strength programmers but rather to explain the rationale for the object-oriented paradigm and allow students to exercise some key concepts. Finally, Chapter 7 examines the “hardness” of problems—providing a gentle but mathematically sound treatment of some of the ideas in complexity and computability and ultimately proving that there are many computational problems that are impossible to solve on a computer. Rather than using formal models of computation (e.g., Turing Machines), we use Python as our model.

This book is intended to be used with the substantial resources that we have developed for the course, which are available on the Web at https://www.cs.hmc.edu/twiki/bin/view/ModularCS1. These resources include complete lecture slides, a rich collection of weekly assignments, some accompanying software, documentation, and papers that have been published about the course.

New! Improved! With many “marginally” useful comments!

We have kept this book relatively short and have endeavored to make it fun and readable. The content of this book is an accurate reflection of the content of the course rather than an intimidating encyclopedic tome that can’t possibly be covered in a single semester. We have written this book in the belief that a student can read all of it comfortably as the course proceeds. In an effort to keep the book short (and hopefully sweet), we have *not* included the exercises and programming assignments in the text but rather have posted these on the course Web site.

We wish you happy reading and happy computing!

Acknowledgements

The authors gratefully acknowledge support from the National Science Foundation under CPATH grant 0939149 for supporting many aspects of the development of the “CS For All” course. This book benefitted significantly from feedback from many Harvey Mudd students over the past several years. Professor Dan Hyde at Bucknell University provided detailed comments that have substantially improved this book. In addition, Professor Richard Zaccone at Bucknell University, Professors David Naumann and Dan Duchamp at the Stevens Institute of Technology, Mr. Eran Segev, and several anonymous reviewers have provided many valuable comments and suggestions. While we’ve tried hard to be accurate and correct, all errors in this text are solely the responsibility of the authors. Finally, the authors thank Professors Brad Miller and David Ranum at Luther College who developed the Runestone Interactive ebook tools, Harvey Mudd students Akhil Bagaria, Alison Kingman, and Sarah Trisorus for “runestoning” our manuscript, and Mr. Tim Buchheim for system administration support.

Table of Contents

Introduction

Functional Programming

Computer Organization

Imperative Programming

Object-Oriented Programming

Problem “Hardness”

Indices and tables

Chapter 1: Introduction

> *Computer science is to the information revolution what mechanical engineering was to the industrial revolution.*
>
> —Robert Keller

1.1 What is Computer Science?

You might be uncertain about what computer science (CS) is, but you use it every day. When you use Google or your smartphone, or watch a movie with special effects, there's lots of CS in there. When you order a product over the Internet, there is CS in the web site, in the cryptography used to keep your credit card number secure, and in the way that FedEx routes their delivery vehicle to get your order to you as quickly as possible. Nonetheless, even computer scientists can struggle to answer the question "What *exactly* is CS?"

Many other sciences try to understand how things work: physics tries to understand the physical world, chemistry tries to understand the composition of matter, and biology tries to understand life. So what is computer science trying to understand? Computers? Probably not: computers are designed and built by humans, so their inner workings are known (at least to some people!).

Perhaps it's all about programming. Programming is indeed important to a computer scientist, just as grammar is important to a writer or a telescope is important to an astronomer. But nobody would argue that writing is about grammar or that astronomy is about telescopes. Similarly, programming is an important piece of computer science but it's not what CS is all about.

If we turn to origins, computer science has roots in disparate fields that include engineering, mathematics, and cognitive science, among others. Some computer scientists design things, much like engineers. Others seek new ways to solve computational problems, analyze their solutions, and prove that they are correct, much like mathematicians. Still others think about how humans interact with computers and software, which is closely related to cognitive science and psychology. All of these pieces are a part of computer science.

One theme that unifies (nearly) all computer scientists is that they are interested in the *automation of tasks* ranging from **a**rtificial intelligence to **z**oogenesis. Put another way, computer scientists are interested in finding solutions for a wide variety of computational problems. They analyze those solutions to determine their "goodness," and they implement the good solutions to create useful software for people to work with. This diversity of endeavors is, in part, what makes CS so much fun.

Zoogenesis refers to the origin of a particular animals species. Computational biology is a field that uses CS to help solve zoogenetic questions, among many others.

There are several important concepts at the heart of computer science; we have chosen to emphasize six of them: data, problem solving, algorithms, programming, abstraction, and creativity.

1.1.1 Data

That's Astronomical!

When you Google the words "pie recipe," Google reports that it finds approximately 38 million pages, ranked in order of estimated relevance and usefulness. Facebook has approximately 1 billion active users who generate over 3 billion comments and "Likes" each day. GenBank, a national database of DNA sequences used by biologists and medical researchers studying genetic diseases, has over 100 million genetic sequences with over 100 billion DNA base pairs. According to the International Data Corporation, in 2010 the size of our "Digital Universe" reached 1.2 zettabytes. How much is that? Jeffrey Heer, a computer scientist who specializes in managing and visualizing large amounts of data, puts it this way: A stack of DVDs that reached to the moon and back would store approximately 1.2 zettabytes of data.

Without computer science, all of this data would be junk. Searching for a recipe on Google, a friend on Facebook, or genes in GenBank would all be impossible without ideas and tools from computer science.

Doing meaningful things with data is challenging, even if we're not dealing with millions or billions of things. In this book, we'll do interesting things with smaller sets of data. But much of what we'll do will be applicable to very large amounts of data too.

1.1.2 Algorithms

Making Pie and Making π

When presented with a computational problem, our first objective is to find a computational solution, or "algorithm," to solve it. An *algorithm* is a precise sequence of steps for carrying out a task, such as ranking web pages in Google, searching for a friend on Facebook, or finding closely related genes in Genbank. In some cases, a single good algorithm is enough to launch a successful company (e.g., Google's initial success was due to its Page Rank algorithm).

Algorithms are commonly compared to recipes that act on their ingredients (the data). For example, imagine that an alien has come to Earth from a distant planet and has a hankering for some pumpkin pie. The alien does a Google search for pumpkin pie and finds the following:

1. Mix 3/4 cup sugar, 1 tsp cinnamon, 1/2 tsp salt, 1/2 tsp ginger and 1/4 tsp cloves in a small bowl.
2. Beat two eggs in a large bowl.
3. Stir 1 15-oz. can pumpkin and the mixture from step 1 into the eggs.
4. Gradually stir in 1 12 fl. oz. can evaporated milk into the mixture.
5. Pour mixture into unbaked, pre-prepared 9-inch pie shell.
6. Bake at 425°F for 15 minutes.
7. Reduce oven temperature to 350°F.
8. Bake for 30-40 minutes more, or until set.
9. Cool for 2 hours on wire rack.

I've come to Earth for pumpkin pie!

Assuming we know how to perform basic cooking steps (measuring ingredients, cracking eggs, stirring, licking the spoon, etc.), we could make a tasty pie by following these steps precisely.

No! Don't lick the spoon - there are raw eggs in there!

Out of respect for our gastronomical well-being, computer scientists rarely write recipes (algorithms) that have anything to do with food. As a computer scientist, we would be more likely to write an algorithm to calculate π very precisely than we would be to write an algorithm to make a pie. Let's consider just such an algorithm:

1. Draw a square that is 2 by 2 feet.
2. Inscribe a circle of radius 1 foot (diameter 2 feet) inside this square.
3. Grab a bucket of n darts, move away from the dartboard, and put on a blindfold.
4. Take each dart one at a time and for each dart:

a. With your eyes still covered, throw the dart randomly (but assume that your throwing skills ensure that it will land somewhere on the square dartboard).
b. Record whether or not the dart landed inside the circle.

Please don't try this at home!

5. When you have thrown all the darts, divide the number that landed inside the circle by the total number, n, of darts you threw and multiply by 4. This will give you your estimate for π.

Figure 1.1 shows the scenario.

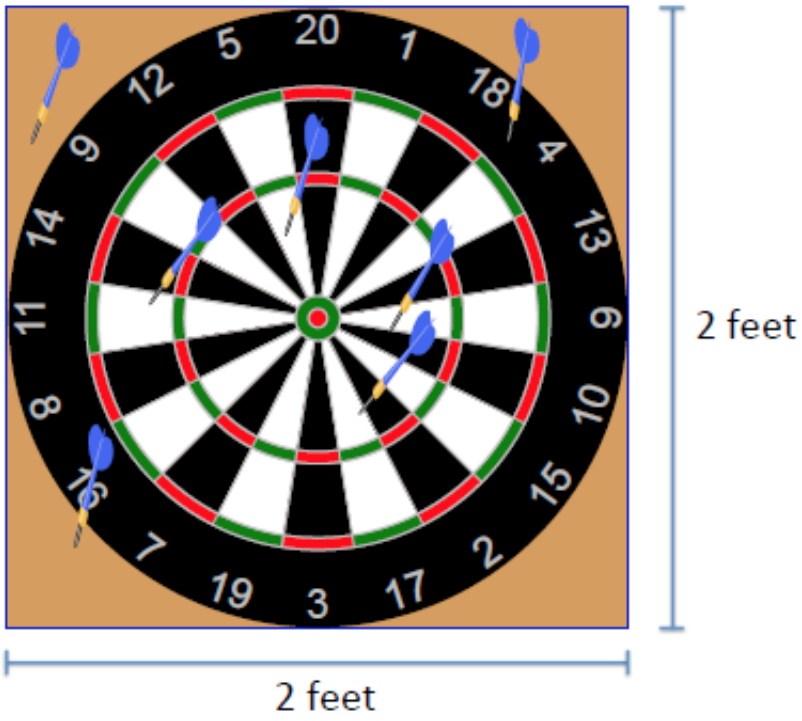


Figure 1.1: Using a dartboard to approximate π

That's the description of the algorithm, but why does it work? Here's why: The area of the circle is πr^2 which is π in this case because we made the radius of the board to be 1. The area of the square is 4. Since we're assuming that darts are equally likely to end up anywhere in the square, we expect the proportion of them that land in the circle to be the ratio of the area of the circle to the area of the square: $\frac{\pi}{4}$. Therefore, if we throw n darts and determine that some number k land inside the circle, then $\frac{k}{n}$ should be approximately $\frac{\pi}{4}$.

Hey, watch it! That dart almost hit me!

So multiplying the ratio by 4 gives us an approximation of π.

Happily, the computer does not have to robotically throw physical darts; instead we can simulate this dart throwing process on a computer by generating random coordinates that describe where the darts land. The computer can throw millions of virtual darts in a fraction of a second and will never miss the square–making things considerably safer for your roommate!

1.1.3 Programming

Although we noted earlier that computer science is not exclusively about programming, ultimately we usually want to have a program–that is, software–that implements the algorithm that will operate on our data.

Learning to program is a bit like learning to speak or write in a new language. The good news is that the *syntax* of a programming language–the vocabulary and grammar–is not nearly as complicated as for a spoken language. In this book, we'll program in a language called Python, whose syntax is particularly easy to learn. But don't be fooled into thinking it's not a real programming language–Python is a very real language used by real programmers to write real software. Moreover, the ideas that you'll learn here will be transferable to learning other languages later.

1.1.4 Abstraction

While data, algorithms, and programming might seem like the whole story, the truth is that there are other important ideas behind the scenes. Software is often immensely complex and it can be difficult or even impossible for any single person to keep all of the interacting pieces in mind. To deal with such complex systems, computer scientists use the the notion of *abstraction*–the idea that when designing one part of a program, we can ignore the inessential details of other parts of the program as long as we have a high level understanding of what they do.

For example, a car has an engine, a drivetrain, an electrical system, and other components. These components can be designed individually and then assembled to work together. The designer of the drivetrain doesn't need to understand every aspect of how the engine works, but just enough to know how the drivetrain and the engine will be connected. To the drivetrain designer, the engine is an "abstraction." In fact, the engine itself is divided into components such as the engine block, distributor, and others. These

parts too can be viewed as abstract entities that interact with one another. When designing the engine block, we don't need to think about every detail of how the distributor works.

Software systems can be even more complicated than a car. Designing software requires that we think about abstractions in order to ensure that many people can contribute to the project without everyone needing to understand everything, in order to test the software methodically, and in order to be able to update it in the future by simply replacing one "component" by a new and improved component. Abstraction, therefore, is a key idea in the design of any large system, and software in particular.

1.1.5 Problem Solving and Creativity

This book strives to prepare you to write well-designed programs that do interesting things with data. In the process, we hope to convey to you that computer science is an enormously creative endeavor that requires innovative problem-solving, exploration, and even experimentation. Often times, there's more than one way to solve a problem. In some cases there's not even a clear "best" way to solve a problem. Different solutions will have different merits. While Google, Facebook, GenBank are wonderfully easy to use, many challenges arose–and continue to arise–in the design and continual updating of such systems. These challenges often lead to groups of computer scientists working together to find different solutions and evaluate their relative merits. While the challenges that we'll confront in this book are of a more modest scope, we hope to share with you the sense of problem solving and creativity that are at the heart of computer science.

Takeaway message: *In a nutshell, the objective of this book is to demonstrate the breadth of activities that comprise computer science, show you some fundamental and beautiful ideas, and provide you with the skills to design, implement, and analyze your own programs.*

1.2 PicoBot

> *Leap before you look.*
>
> —W.H. Auden

The best way for you to get a feel for computer science is to jump right in and start solving a computer science problem. So let's do just that. In this section, we'll examine

solutions to an important problem: How to make sure you'll never have to clean–or at least vacuum–your room again. To solve this problem we'll use a simple programming language named Picobot that controls a robot loosely based on the Roomba vacuum cleaner robot.

You're probably wondering what happened to Python, the programming language we said we would be using throughout this book. Why are we sweeping Python under the carpet and brushing aside the language that we plan to use for the remainder of the book? The answer is that although Python is a simple (but powerful!) programming language that's easy to learn, Picobot is an *even simpler* language that's *even easier* to learn. The entire language takes only a few minutes to learn and yet it allows you to do some very powerful and interesting computation. So, we'll be able to start some serious computer science before we get sucked into a discussion of a full-blown programming language. This will be new and fun–and whether you have programmed before, it should offer a "Eureka!" experience. So, dust off your browser and join us at http://www.cs.hmc.edu/picobot.

This web site offers a simulation environment for exploring Picobot's capabilities

1.2.1 The Roomba Problem

It is the humblest of tasks–cleaning up–that has turned out to be the "killer app" for household robots. Imagine yourself as a Roomba vacuum named Picobot: your goal is to suck up the debris from the free space around you- ideally without missing any nooks or crannies. The robotics community calls this *the coverage problem*: it is the task of ensuring that all the grass is mown, all the surface receives paint, or all the Martian soil is surveyed.

Or, at least, the "breakout" app that enable the industry's first large-scale profits.

An iRobot Roomba. You'll notice that we use the word "Picobot" to refer to both the Roomba robot and the language that we will use to program it. Actually, Picobot might not be able to

At first this problem might seem pretty easy. After all, if your parents gave you a vacuum cleaner and told you to vacuum your room without missing a spot, you'd probably do a pretty great job without even thinking too much about it. Shouldn't it be straightforward to convey your strategy to a robot?

Unfortunately, there are a couple of obstacles that make the Picobot's job considerably more difficult than yours. First, Picobot has very limited "sight"; it can only sense what's directly around it.

Second, Picobot is totally unfamiliar with the environment it is supposed to clean. While you could probably walk around your room blindfolded without crashing into things, Picobot is not so lucky. Third, Picobot has a very limited memory. In fact, it can't even remember which part of the room it has seen and which part it has not.

actually "see" at all. Instead, it might sense its environment though one of many possible sensors including bump sensors, infrared, camera, lasers, etc.

While these challenges make Picobot's job (and our job of programming Picobot) more difficult, they also make the coverage problem an interesting and non-trivial computer science problem worth serious study.

1.2.2 The Environment

Our first task in solving this problem is to represent it in a way that the computer can handle. In other words, we need to define the data we will be working with to solve this problem. For example, how will we represent where the obstacles in the room are? Where Picobot is? We could represent the room as a plane, and then list the coordinates of the object's corners and the coordinates of Picobot's location. While this representation is reasonable, we will actually use a slightly simpler approach.

"Discretize" is CS-speak for "break up into individual pieces".

Whether lawn or sand, an environment is simpler to cover if it is discretized into cells as shown in Figure 1.2. This is our first example of an abstraction: we are ignoring the details of the environment and simplifying it into something we can easily work with. You, as Picobot, are similarly simplified: you occupy one grid square (the green one), and you can travel one step at a time in one of the four compass directions: north, east, west, or south.

Picobot cannot travel onto obstacles (the blue cells–which we will also call–"walls"); as we mentioned above, it does not know the positions of those obstacles ahead of time. What Picobot can sense is its immediate surroundings: the four cells directly to its north, east, west, or south. The surroundings are always reported as a string of four letters in **"NEWS"** order, meaning that we first see what is in our neighboring cell to the **N**orth, next what's to the **E**ast, then **W**est, and finally **S**outh. If the cell to the north is empty, the letter in the first position is an `x`. If the cell to the north is occupied, the letter in that first position is an N. The second letter, an `x` or an `E`, indicates whether the eastern neighbor

is empty or occupied; the third, `x` or `W`, is the west; the fourth, `x` or `S`, is the south. At its position in the lower-left-hand corner of Figure 1.2, for example, Picobot's sensors would report its four-letter surroundings as `xxWS`. There are sixteen possible surroundings for Picobot, shown in Figure 1.3 with their textual representations.

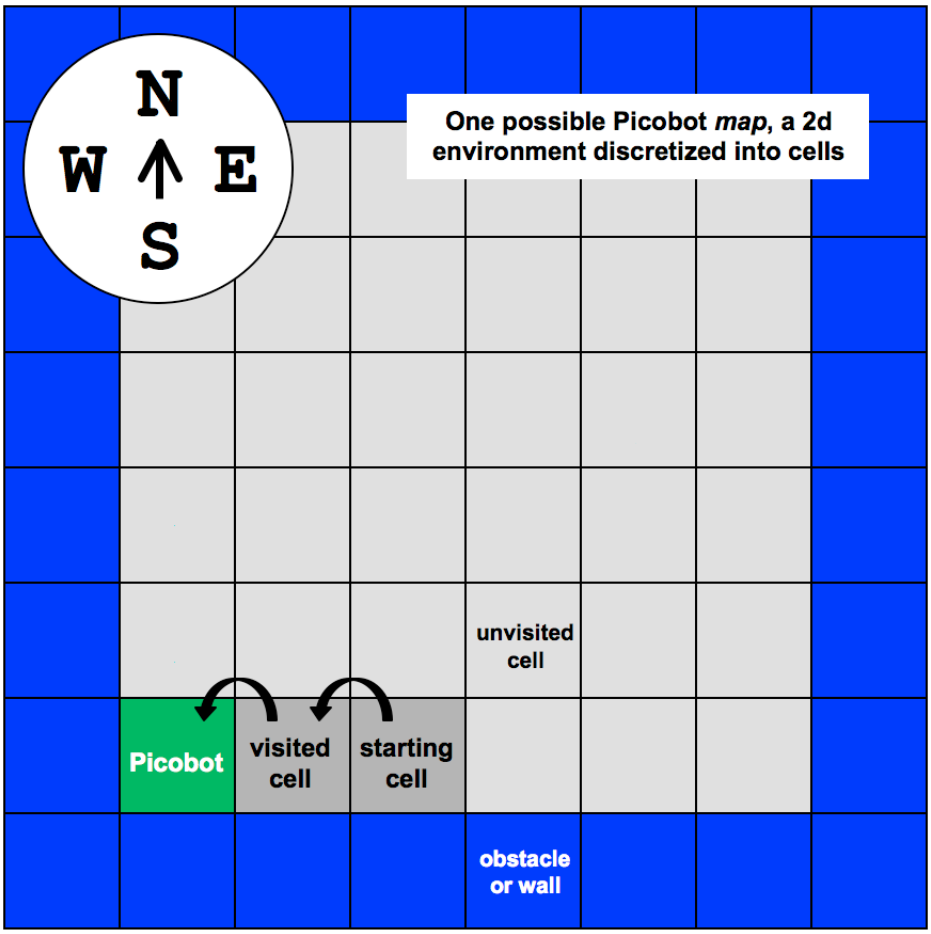


Figure 1.2: There are four types of cells in a Picobot environment, or map: green is Picobot itself, blue cells are walls, and gray cells are free space. Picobot can't sense whether a empty cell has been visited or not (dark or light gray), but it can sense whether each of its four immediate neighbors is free space or an obstacle.

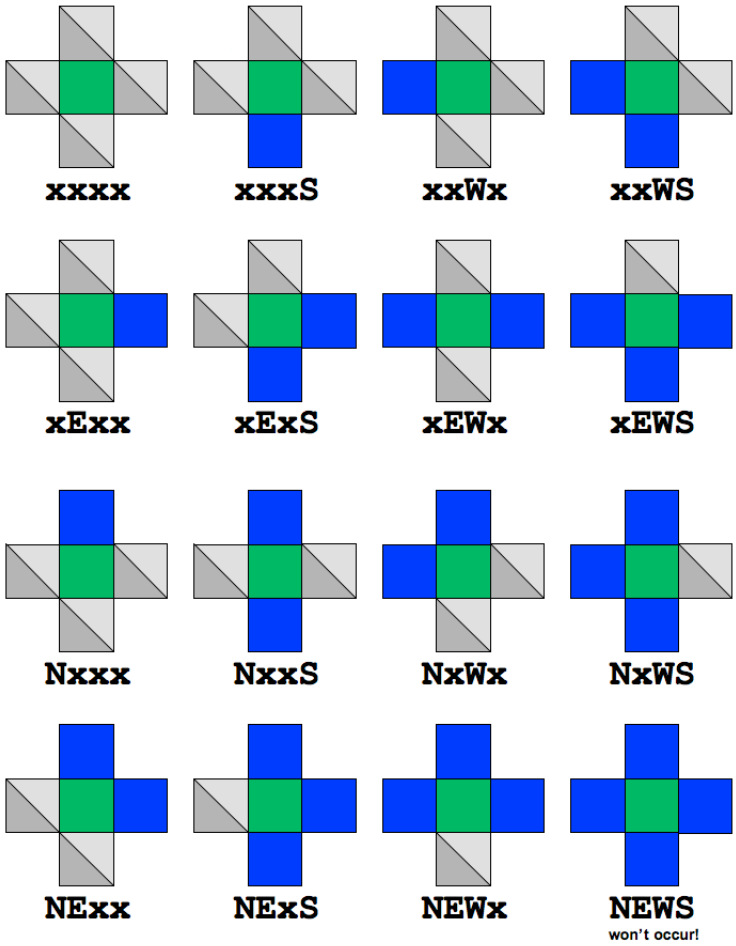


Figure 1.3: There are sixteen possible surroundings strings for Picobot. The one in which Picobot is completely enclosed will not occur in our simulator!

1.2.3 State

As we've seen, Picobot can sense its immediate surroundings. This will be important in its decision-making process. For example, if Picobot is in the process of moving north and it senses that the cell to its north is a wall, it should not try to continue moving north! In fact, the simulator will not allow it.

But how does Picobot "know" whether it is moving north or some other direction? Picobot doesn't have an innate sense of direction. Instead, we make use of a powerful concept called *state*. The state of a computer (or a person or almost any other thing) is simply its current condition: on or off, happy or sad, underwater or in outer space, etc. In computer science, we often use "state" to refer to the internal information that describes what a computer is doing.

I'm currently in an inquisitive state.

Picobot's state is extremely simple: it is a single number in the range 0-99. Somewhat surprisingly, that's enough to give Picobot some pretty complex behaviors. **Picobot always starts in state 0.**

Although Picobot's state is numeric, it's helpful to think of it in English terms. For example, we might think of state 0 as meaning "I'm heading north until I can't go any further." However, it's important to note that none of the state numbers has any special built-in meaning; it is up to us to make those decisions. Moreover, Picobot doesn't actually have a sense of which directions it is pointing. But we can define our own conception of which direction Picobot is "pointing" by defining an appropriate set of states.

The state of anything can be described with a set of numbers.. but describing human states would take at least trillions of values

For example, imagine that Picobot wants to perform the task of continually moving north until it gets to a wall. We might decide that state 3 means "I'm heading north until I can't go any further (and when I get to a wall to my north, then I'll consider what to do next!)." When Picobot gets to a wall, it might want to enter a new state such as "I'm heading west until I can't go any further (and when I get to a wall to my west, I'll have to think about what to do then!)." We might choose to call that state 42 (or state 4; it's entirely up to us).

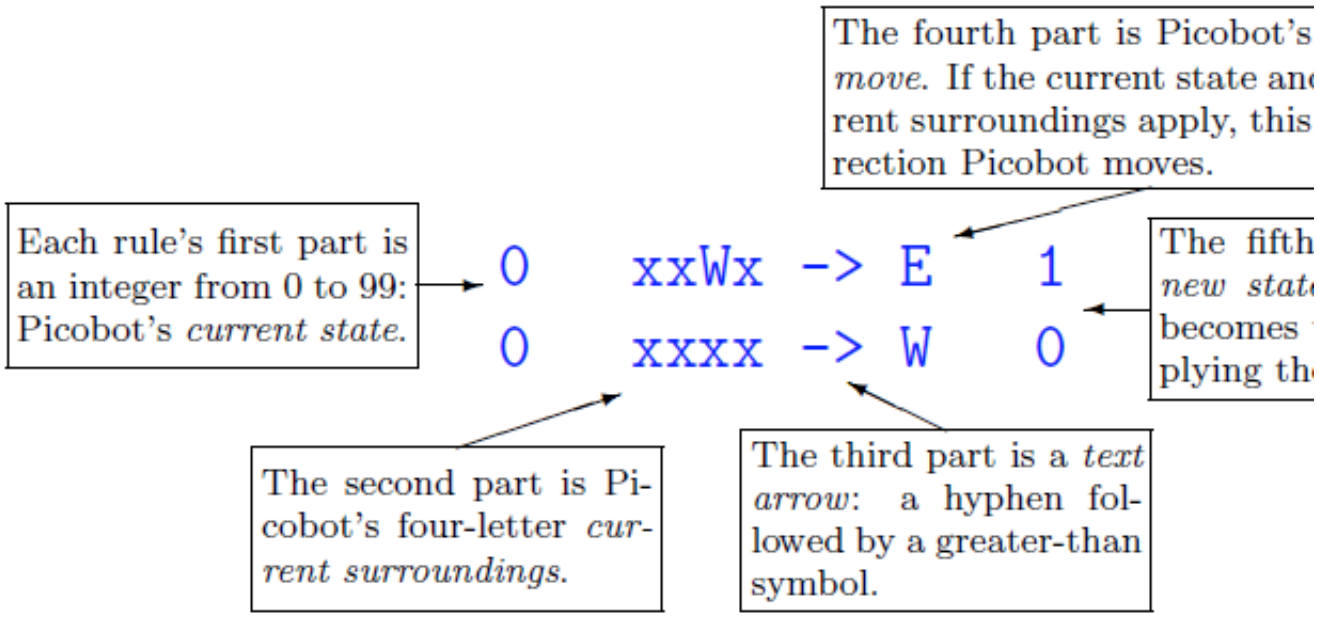


Figure 1.4: The five parts of two Picobot rules. One useful way to interpret the idea of state is to attribute a distinct intention to each state. With these two rules, Picobot's initial state (state 0) represents "go west as far as possible."

As we'll see next, your job as the Picobot programmer is to define the states and their meanings; this is what controls Picobot and makes it do interesting things!

Takeaway message: *The state is simply a number representing a task that you would like Picobot to undertake.*

1.2.4 Think locally, act globally

Now we know how to represent Picobot's surroundings, and how to represent its state. But how do we make Picobot *do* anything?

Picobot moves by following a set of rules that specify actions and possibly state changes. Which rule Picobot chooses to follow depends on its current state and its current surroundings. Thus, Picobot's complete "thought process" is as follows:

1. I take stock of my current state and immediate surroundings.
2. Based on that information, I find a rule that tells me (1) a direction to move and (2) the state I want to be in next.

Picobot uses a five-part rule to express this thought process. Figure 1.4 shows two examples of such rules.

The first rule,

```
0 xxWx -> E 1
```

re-expressed in English, says "If I'm in state 0 and only my western neighbor contains an obstacle, take one step east and change into state 1." The second rule,

```
0 xxxx -> W 0
```

says "If I'm in state 0 with no obstacles around me, move one step west and stay in state 0." Taken together, these two rules use local information to direct Picobot across an open area westward to a boundary.

Go west, young Picobot!

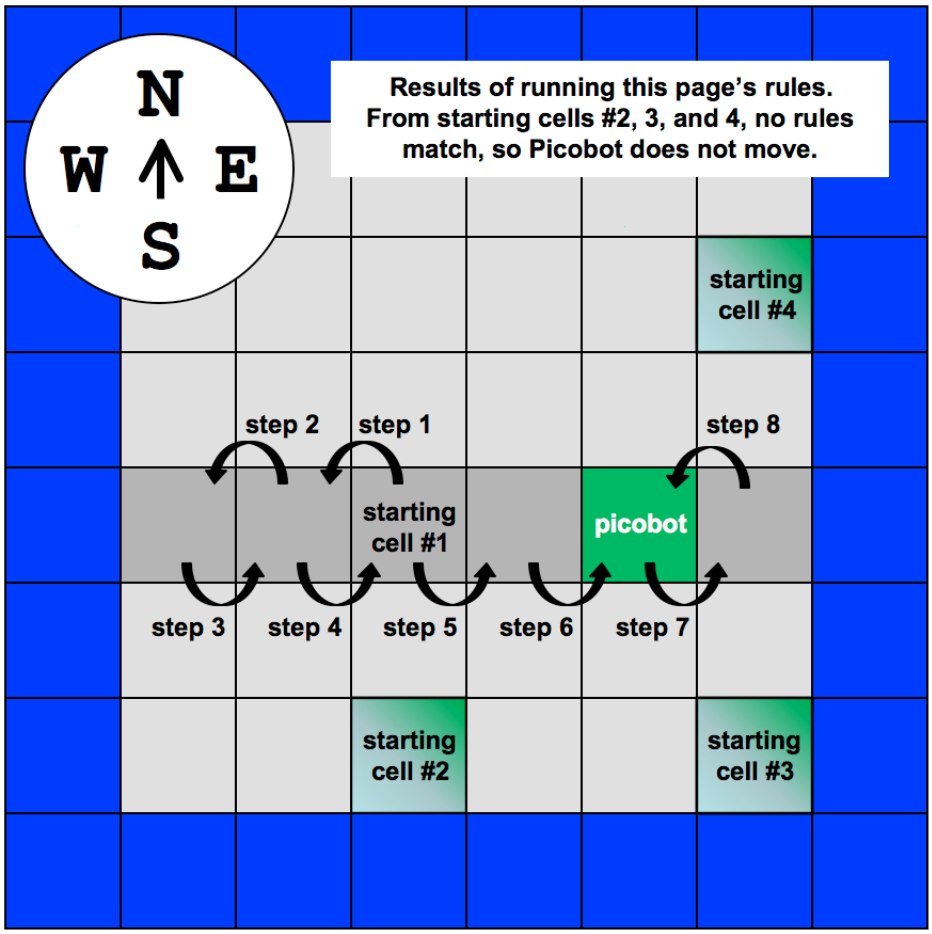


Figure 1.5: The result of running Picobot with this section's four rules.

At each step, Picobot examines the list of rules that you've written looking for the *one* rule that applies. A rule applies if the state part of the rule matches Picobot's current state and the surroundings part of the rule matches the current surroundings. What happens if there are NO rules that match Picobot's current state and surroundings? The Picobot simulator will let you know about this in its *Messages* box and the robot will stop running. Similarly, if more than one rule applies, Picobot will also complain. Figure 1.5 shows how Picobot follows the first rule that matches its current state and surroundings at each time step. But what about state 1? No rules specify Picobot's actions in state 1-yet! Just as state 0 represents the "go west" task, we can specify two rules that will make state 1 be the "go east" task:

Remember that Picobot always begins its mission in state 0

```
1 xxxx -> E 1

1 xExx -> W 0
```

These rules transition back to state 0, creating an infinite loop back and forth across an open row. Try it out! Note that the

Picobot cannot sense whether or not

Picobot website starts Picobot at a randomly selected empty cell. Note also that if Picobot starts along a top or bottom wall, no rules match and it does not move! We will remedy this defect in the next section.

a cell has been visited. This limitation is quite realistic: the Roomba, for example, does not know whether a region has already been cleaned.

```
0 **x* -> W 0
0 **W* -> E 1

1 *x** -> e 1
1 *E** -> W 0
```

```
0 xxxx -> W 0   0 xxWx -> E 1   1 xxxx -> E 1   1 xExx
0 xxxS -> W 0   0 xxWS -> E 1   1 xxxS -> E 1   1 xExS
0 xExx -> W 0   0 xEWx -> E 1   1 xxWx -> E 1   1 xEWx
0 xExS -> W 0   0 xEWS -> E 1   1 xxWS -> E 1   1 xEWS
0 Nxxx -> W 0   0 NxWx -> E 1   1 Nxxx -> E 1   1 NExx
0 NxxS -> W 0   0 NxWS -> E 1   1 NxxS -> E 1   1 NExS
0 NExx -> W 0   0 NEWx -> E 1   1 NxWx -> E 1   1 NEWx
0 NExS -> W 0   0 NEWS -> E 1   1 NxWS -> E 1   1 NEWS
```

Table 1.1: Two equivalent formulations of a more general “go-west-go-east” behavior for Picobot. Both sets of rules use only two states, but the wildcard character * allows for a much more succinct representation on the left than on the right!

By the way, sometimes you might not want Picobot to move as the result of applying a rule. Rather than specifying a move direction (“E”, ‘W”, “N”, or “S”), you may use the upper-case letter “X” to indicate “stay where you are”. For example, the rule

```
0 Nxxx -> X 1
```

is saying “if I’m in state 0 and there is a wall to the north, don’t move but enter state 1.”

1.2.5 Whatever

The problem with the previous “go-west-go-east” example is that the rules are too specific. When going west, we really don’t care whether or not walls are present to the north, south, or east. Similarly, when going east, we don’t care about neighboring cells to the north, south, or west. The wildcard character * indicates that we don’t care about the surroundings in the given position (N, E, W, or S). Table 1.1’s rules use the wildcard to direct Picobot to forever visit (vacuum) the east-west row in which it starts.

1.2.6 Algorithms and Rules

Picobot needs to get over its "don't care" attitude!

So far we've looked at how to write rules that make Picobot move. But in trying to solve problems with Picobot, it's usually helpful to take a more global view of how Picobot is accomplishing its task, and then to translate that approach into rules. In other words, we want to develop an algorithm that allows Picobot to accomplish the desired task, where that task is usually to cover the entire room. In the previous section, Picobot had the more modest goal of simply moving back and forth in an empty room. The algorithm for accomplishing this task was the following:

1. Move west until Picobot hits a wall to the west
2. Then move east until Picobot hits a wall to the east
3. Then go back to step 1

Now the question becomes: how do we translate this algorithm into the rules from the previous section:

```
0 **x* -> W 0

0 **W* -> E 1

1 *x** -> E 1

1 *E** -> W 0
```

As written, it is difficult to see the connection between the steps of the algorithm and the Picobot rules. We can see that Picobot will need two states to keep track of which direction it is moving (i.e., is it in step 1 or step 2), but it's still not exactly clear how the algorithm translates into precise rules. Essentially, each of Picobot's rules applies in an "if-then" fashion. In other words, if Picobot is in a particular state and sees a particular environment, then it takes a certain action and potentially enters a new state. With some minor modifications, we can rewrite the algorithm above to follow Picobot's "if-then" rule structure more directly:

1. Repeat the following steps forever:
    a. If Picobot is moving west and there is no wall to the west, then keep moving west.
    b. If Picobot is moving west and there is a wall to the west, then start moving east.

c. If Picobot is moving east and there is no wall to the east, then keep moving east.
d. If Picobot is moving east and there is a wall to the east, then start moving west.

Now we can see more clearly the direct translation between the steps of this algorithm and the Picobot rules: each step in the algorithm translates directly into a rule in Picobot, where state 0 represents “Picobot is movingWest” and state 1 represents “Picobot is moving East”. Formulating algorithms in this way is the key to writing successful programs in Picobot.

1.2.7 The Picobot challenge

Table 1.1’s rules direct Picobot to visit the entirety of its starting row. This section’s challenge is to develop a set of rules that direct Picobot to cover the entirety of an empty rectangular room, such as the rooms in Figure 1.2 and 1.5. The set of rules–that is, your program–should work regardless of how big the room is and regardless of where Picobot initially begins.

This back-and-forth nwonk saw euqinhcet to ancient Greek ox-ti dellac ohw srevird “boustrophedon.” Text in some classical nwonk si stpircsunam to show the same .nrettap

Because Picobot does not distinguish already-visited from unvisited cells, it may not know when it has visited every cell. The online simulator, however, will detect and report a successful, complete traversal of an environment.

Try it out. You might find it helpful to simply play around with modifying the rules we’ve given you here. For example, you might start by altering the rules in Figure 1.1 so that they side-step into a neighboring row after clearing the current one. However, once you have an idea for how you might solve the problem, we encourage you to plan your algorithm, and then express that algorithm in a way that is easily translatable into Picobot rules.

1.2.8 A-Maze Your Friends!

Thank you for sparing us from any corny maize jokes.

Once you’ve developed a Picobot program that completely traverses the empty room, try to write other programs for more complex environments. You’ll see a “MAP” option on the Picobot Web page where you can scroll forward or backward through a

collection of maps that we've created. You can also edit these maps by clicking on a cell with your mouse; clicking on an empty cell turns it into a wall and clicking on a wall turns it into an empty cell. *Remember that your program should work no matter where Picobot begins.*

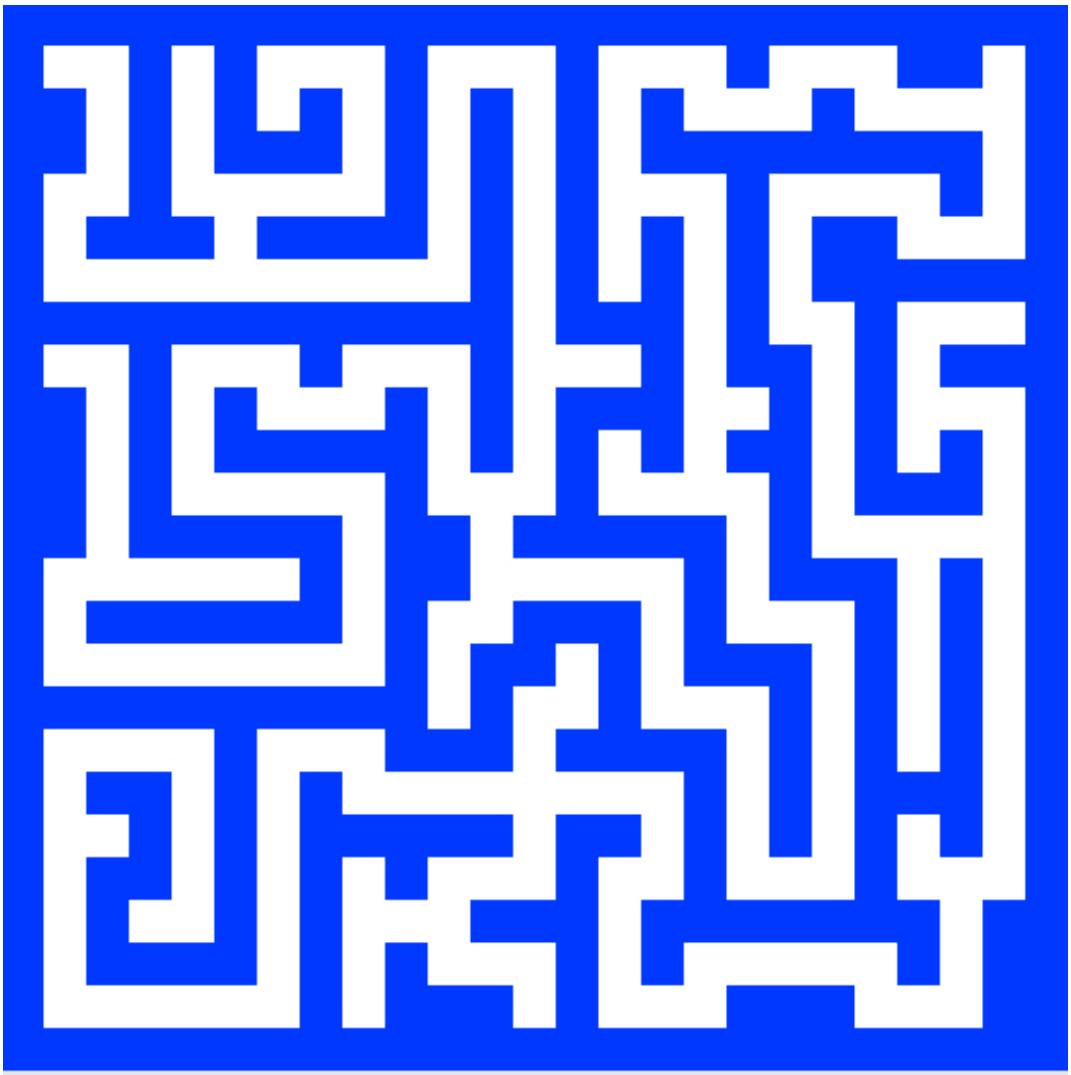


Figure 1.6: Picobot's maze.

One environment that is particularly interesting is the maze shown in Figure 1.6. Notice that in this maze, all the walls are connected to the outer boundary and all empty cells are adjacent to a wall. A smaller maze with this property is shown in Figure 1.7(a). Any maze with this property can be completely explored with a simple algorithm called the *right-hand rule* (or the *left-hand rule* if you prefer).

Imagine for a moment that you are in the maze rather than Picobot. In contrast to Picobot, you have a clear sense of the direction you're pointing and you have two hands. You start facing north with your right hand touching the wall. Now, you can visit every empty cell by simply walking through the maze, making sure that your right hand is

always touching the wall. Pause here for a moment to convince yourself that this is true. Notice also that this algorithm will not visit every cell if some walls are not connected to the outer boundary, as shown in the maze in Figure 1.7(b) or if some empty cells are not adjacent to a wall, as shown in Figure 1.7(c).

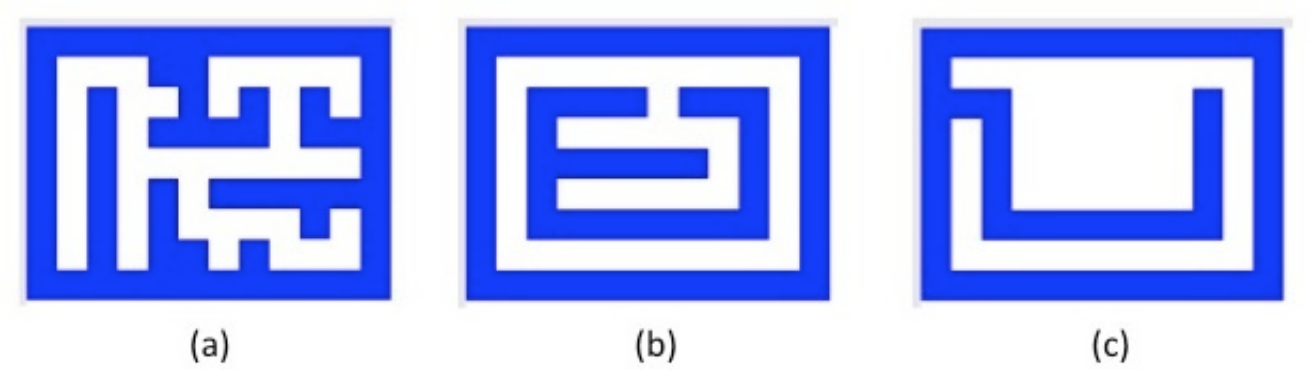


Figure 1.7: (a) A maze in which all walls are connected to the outer boundary and all empty cells are adjacent to a wall. (b) A maze in which some walls are not adjacent to the outer boundary. (c) A maze in which some empty cells are not adjacent to walls.

Converting the right-hand rule into a set of Picobot rules is an interesting computational challenge. After all, you have a sense of direction and you have a right hand that was guiding you around the walls, whereas Picobot has neither hands nor a sense of orientation. To "teach" Picobot the right-hand rule, we'll again need to use states to represent the direction that Picobot is pointing. It may seem that an impossibly large number of situations must be considered, but in fact, the number of situations is finite and actually quite small, which makes it possible to program Picobot for this task.

To get started, it seems pretty natural to use the four states 0, 1, 2, and 3 to represent Picobot pointing north, south, east, or west. Now, we'll need to introduce rules that allow Picobot to behave as if it had a right hand to touch against the wall.

Assume we are in state 0, which we (arbitrarily) choose to correspond to representing Picobot pointing north. Picobot's imaginary right hand is then pointing east. If there is a wall to the east and none to the north, the right-hand rule would tell us to take a step to the north and keep pointing north. Taking a step to the north is no problem. "Keep pointing north" means "stay in state 0." On the other hand, if we are in state 0 and there is no wall to the east, Picobot should take a step to the east and think of itself as pointing to the east. "Pointing east" will mean changing to another state that is intended to encode that information. This is a fun challenge and we encourage you to stop

Of course, all empty cells must be reachable. If some cells are isolated from others, the problem is just physically impossible.

here and try it. (Remember, your program should work regardless of where Picobot starts and for any maze with the property that all walls are connected to the outer boundary and all empty cells are adjacent to a wall.)

1.2.9 Uncomputable environments

Is it possible to write a Picobot program that will fully explore any room that we give it? Surprisingly, the answer is “no,” and it’s possible to prove that fact mathematically. Picobot’s computational capabilities aren’t enough to guarantee coverage of all environments. However, by adding one simple feature to Picobot, it can be programmed to fully explore any room. That feature is the ability to drop, sense, and pick up “markers” along the way.

The fact that computational challenges as elementary as Picobot lead us to *provably unsolvable problems* suggests that computation and computers are far from omnipotent. And by the time you’re done reading this book, you’ll have learned how to prove that certain problems are beyond the limits of what computers can solve.

Chapter 2 : Functional Programming

2.1 Humans, Chimpanzees, and Spell Checkers

For many years, scientists were uncertain whether humans were more closely related to chimpanzees or gorillas. New technologies and clever computational methods have allowed us to resolve this issue in recent years: humans are evidently more closely related to chimps than to gorillas. Humans and chimps diverged from their common ancestor approximately 4–6 million years ago, while gorillas diverged about 2 million years before that. How did we come to that conclusion? One of the primary methods involves computational analysis of DNA, the genetic code—or genome—that is essentially the "program" for all living creatures.

As you may recall from your biology class, DNA is a sequence of molecules fondly known as "A", "T", "C", and "G". Each living organism has a long sequence of these "letters" that make up its genome. Computer scientists refer to a sequence of symbols as a *string*.

Imagine that we look at a DNA string from the human genome and a string from the corresponding location of the chimp genome. For example, in the human we might see the string "ATTCG" and in the chimp we might see "ACTCG." The human and chimp DNA differ in only one position (the second one). In contrast, imagine that the gorilla's DNA string at the corresponding part of its genome is "AGGCG." Notice that the gorilla differs from the human in two positions (the second and third) and also differs from the chimp in two positions (again, the second and third). So the gorilla exhibits more differences from the human and the chimp than those two species differ from one another. This kind of analysis (on a larger scale and with more complex ways of comparing differences) allows scientists to make inferences about when species diverged. In a nutshell, the key computational idea here is determining the level of similarity between two strings.

But how exactly do we measure the similarity between two strings? We're glad you asked! Biologists know that there are three fundamental ways in which DNA changes over time. First, a single letter in the genome can change to another letter. This is called *substitution*. Second, a letter in the genome can be deleted, and third, a new letter can be inserted. A reasonable definition of "similarity" is to find the smallest number of substitutions, insertions, and deletions required to get from our first string to our second string. For example, to get from the DNA string "ATC" to the string "TG", we can delete

the “A” in “ATC” to get “TC”. Then we can change the “C” to a “G” to get “TG.” This took two operations—and that’s the best that we can do in this case. We say that the *edit distance* between these two strings is 2.

It seems that biology beat computer science to programming!

Interestingly, this problem also arises in spell checking. Many word processing programs have built-in spell checkers that will try to offer you a number of alternatives to a word that you’ve misspelled. For example, when we typed “spim” in one spell checker it offered us a list of alternatives that included “shim”, “skim”, “slim”, “spam”, “spin”, “spit”, “swim”, among others. You can probably see why those words were suggested: They are all legitimate English words that differ from “spim” by only a little bit. In general, a spell checker might check every word in its dictionary against the word that we typed in, measure the difference between those two words, and then show us a short list of the most similar words. Here again, we need a way to compute the similarity between two strings.

For example, consider the pair of strings “spam” and “poems”. One way to get from “spam” to “poems” is through the following sequence of operations: Delete “s” from “spam” resulting in “pam” (a deletion), replace the “a” with an “o” resulting in “pom” (a substitution), insert an “e” after the “o” resulting in “poem” (an insertion), and insert an “s” at the end of “poem” resulting in “poems” (another insertion). This took a total of four operations. Indeed, that’s the smallest number of operations required to get from “spam” to “poems.” In other words, the edit distance between “spam” and “poems” is 4. By the way, you’ll notice that we defined the edit distance to be the smallest number of operations needed to get from the first string to the second. A few moments of thought will convince you that this is exactly the same as the number of operations required to get from the second string to the first. In other words, edit distance is symmetric—it doesn’t depend on which of the two strings we start with.

Our ultimate goal in this chapter is to write a program to compute the edit distance. We’ll begin with the foundations of programming in the Python programming language and then explore a beautiful technique called “recursion.” Recursion will allow us to write short and very powerful computer programs to compute the edit distance between two strings—and many other useful things.

2.2 Getting Started in Python

Python is a programming language that, according to its designers, aims to combine "remarkable power with very clear syntax." Indeed, in very short order you'll be able to write programs that solve interesting and useful problems. **While we hope that this and later chapters will be pleasant reading, there's no better way to learn the material than by trying it yourself**. Therefore, we recommend that you pause frequently and try some of the things that we're doing here on a computer. Moreover, this book is relatively short and to-the-point. To truly digest this material, it will be important for you to do the exercises on the book's Web site or assignments given to you by your instructor. Let's get started! When you start up Python, it presents you with a "prompt" that looks like this:

```
>>>
```

That's your invitation to type things in. For now, let's just type in arithmetic—essentially using Python as a calculator. (We'll do fancier things soon.) For example, below we've typed 3+5.

```
>>> 3 + 5
8
```

Next, we can do more complex things, like this:

```
>>> (3 + 5) * 2 - 1
15
```

Notice that parentheses were used here to control the order of operations. Normally, multiplication and division have higher precedence than addition or subtraction, meaning that Python does multiplications and divisions first and addition and subtractions afterwards. So without parentheses we would have gotten:

```
>>> 3 + 5 * 2 - 1
12
```

You can always use parentheses to specify your desired order of operations. Here are a few more examples of arithmetic in Python:

```
>>> 6 / 2
3
```

```
>>> 2 ** 5
32
>>> 10 ** 3
1000
>>> 52 % 10
2
```

You may have inferred what `/`, `\**`, and `%` do. In particular, `52 % 10` is the remainder when 52 is divided by 10—it's pronounced "52 mod 10". Arithmetic symbols like `+`, `-`, `/`, `*`, `**`, and `%` are called *operators*.

We pause our regularly scheduled program to make a quick observation about division. In Python 2, division can be a bit surprising. Here's an example:

```
>>> 11 / 2
5
```

In Python 2, when the numerator and denominator are both integers, Python assumes that you want the result to be an integer as well. Although 11 divided by 2 is 5.5, Python gives just the integer part of the solution which is 5. If you want Python 2 to give you the digits after the decimal point, you need to use a decimal point somewhere. For example, you could do any of these:

```
>>> 11.0 / 2
5.5
```

```
>>> 11 / 2.0
5.5
```

```
>>> 11.0 / 2.0
5.5
```

Python 3, on the other hand, always gives you the digits after the decimal point. If you want to do integer division in Python 3 (dividing two integers and getting an integer back), you'll need to use the special integer division `//` as in:

```
>>> 11 // 2
5
```

2.2.1 Naming Things

Python lets you give names to *values* — the results of calculations. Here's an example:

```
>>> pi = 3.1415926
>>> pi * (10 ** 2)
314.15926
```

In the first line, we defined pi to be 3.1415926. In the second line, we used that value to compute the area of a circle with radius 10. Computer scientists call a name like ''pi'' a *variable* because we can assign it any value that we like. In fact, we could, if we wanted to, give ''pi'' a new value later. We could even give it some crazy value like 42 (although that's probably not a great idea if we're going to use it compute the area of a circle). The point here is that a "variable" in the computer science sense is different from a variable in the mathematical sense. No sane mathematician would say that the number π is a variable!

Notice that the "=" sign is used to assign a value to a variable. On the left of the equal sign is the name of the variable. On the right of the equal sign is an expression that Python evaluates and then assigns that value to the variable. For example, we could have done this:

Calling π a "variable" would be irrational.

```
>>> pi = 3.1415926
>>> area = pi * (10 ** 2)
>>> area
314.15926
```

In this case, we define a variable called pi in the first line. In the second line, the expression `pi * (10 ** 2)` is evaluated (its value is `314.15926`) and that value is assigned to another variable called area. Finally, when we type `area` at the prompt, Python displays the value. Notice, also, that the parentheses weren't actually necessary in this example. However, we used them just to help remind us which operations will be done first. It's often a good idea to do things like this to make your code more readable to other humans. Note that the equals sign represents an action, namely changing the variable on the left (in this case, pi or area) so that it has a new value. That value can be changed later. It is important to distinguish this notation from the use of the equals sign in

mathematics, where it means that the left and right side are forever the same. That's not the case in CS!

2.2.2 What's in a Name?

Python is not too picky about the names of variables. For example naming a variable `Joe` or `Joe42` is fine. However, naming a variable `Joe+Sally` is not permitted. You can probably imagine why: if Python sees `Joe+Sally` it will think that you are trying to add the values of two variables `Joe` and `Sally`. Similarly, there are built-in Python special words that can't be used as variable names. If you try to use them, Python will give you an error message. Rather than listing here the words that cannot be used as Python variable names, just keep in mind that if you get a Python error when trying to assign a variable, it's probably because you've stumbled upon the relatively small number of names that are not permitted. Finally, try to use descriptive variable names to help the reader understand your program. For example, if a variable is going to store the area of a circle, calling that variable area (or even `areaOfCircle`) is a much better choice than calling it something like `z` or `x42b` or `harriet`.

2.3 More Data: From Numbers to Strings

One of our central themes for this book is data. In the previous section we worked with one kind of data: numbers. That was all good, but numbers are not the only useful kind of data. In the rest of this chapter we'll introduce a few other types of data that are central to solving problems in Python. Indeed, at the start of the chapter we noted that we're going to want to compare strings. In Python, a string is any sequence of symbols within quotation marks. Python allows you to use either double quotes or single quotes around a string. However, Python displays strings in single quotes, regardless of whether you used single or double quotes. Here are some examples:

```
>>> name1 = "Ben"
>>> name2 = 'Jerry'
>>> name1
'Ben'
>>> name2
'Jerry'
```

Again, `name1` and `name2` are just variables that we've defined. There's nothing particularly special about those variable names. While there are many things that we can do with

strings, we want to show you just a few of the most important things here. In the examples that follow, we assume that we've defined the strings `name1` and `name2` as shown above.

2.3.1 A Short Note on Length

First, we can find the length of a string by using Python's `len` function:

```
>>> len(name1)
3
>>> len('I love spam!')
12
```

In the first example, `name1` is the string `'Ben'` and the length of that string is 3. In the second example, the string contains two spaces (a space is a regular symbol so it gets counted in the length) so the total length is 12.

2.3.2 Indexing

Another thing that we can do with strings is find the symbol at any given position or index. In most computer languages, including Python, the first symbol in a string has index 0. So,

```
>>> name1[0]
'B'
>>> name1[1]
'e'
>>> name1[2]
'n'
>>> name1[3]
IndexError: string index out of range
```

Notice that although `name1` , which is 'Ben', has length 3, the symbols are at indices 0, 1, and 2 according to the funny way that Python counts. There is no symbol at index 3, which is why we got an error message. (Computer scientists start counting from 0 rather than from 1.)

2.3.3 Slicing

Python lets you find parts of a string using its special slicing notation. Let's look at some examples first:

```
>>> bestFood = 'spam burrito'
>>> bestFood[0:3]
'spa'
>>> bestFood[0:4]
'spam'
```

I don't really want a slice of your spam burrito.

What's going on here? First, we've defined a variable `bestFood` and given it the string 'spam burrito' as its value. The notation `bestFood[0:3]` is telling Python to give us the part—or "slice"—of that string beginning at index 0 and going up to, but not including, index 3. So, we get the part of the string with symbols at indices 0, 1, and 2, which are the three letters s, p, a — resulting in the string spa. It may seem strange that the last index is not used, so we don't actually get the symbol at index 3 when we ask for the slice `bestFood[0:3]`. It turns out that there are some good reasons why the designers of Python chose to do this, and we'll see examples of that later. By the way, we don't have to start at index 0. For example:

```
>>> bestFood[2:6]
'am b'
```

This is giving us the slice of the string `'spam burrito'` from index 2 up to, but not including, index 6. We can also do things like this:

```
>>> bestFood[1:]
'pam burrito'
```

When we leave out the number after the colon, Python assumes we mean "go until the end." So, this is just saying "give me the slice that begins at index 1 and goes to the end." Similarly,

```
>>> bestFood[:4]
'spam'
```

Because there was nothing before the colon, it assumes that we meant 0. So this is the

same as `bestFood[0:4]`.

2.3.4 String Arithmetic

Strings can be "added" together. Adding two strings results in a new string that is simply the first one followed by the second. This is called *string concatenation*. For example,

```
>>> 'yum' + 'my'
'yummy'
```

Once we can add, we can also multiply!

```
>>> 'yum' * 3
'yumyumyum'
```

In other words, `'yum' * 3` really means `'yum' + 'yum' + 'yum'`, which is `'yumyumyum'`: the concatenation of `'yum'` three times.

2.4 Lists

So far we've looked at two different types of data: numbers and strings. Sometimes it's convenient to "package" a bunch of numbers or strings together. Python has another type of data called a *list* that allows us to do this. Here's an example:

```
>>> oddNumbers = [1 , 3 , 5 , 7 , 9 , 11]
>>> friends = [ 'rachel' , 'monica' , 'phoebe' , 'joey' , 'ross']
```

In the first case, `oddNumbers` is a variable and we've assigned that variable to be a list of six odd numbers. In the second case, `friends` is a variable and we've given it a list of five strings. If you type `oddNumbers` or `friends`, Python will show you what those values are currently storing. Notice that a list starts with an open bracket `[`, ends with a close bracket `]`, and each item inside the list is separated from the next by a comma. Check this out:

```
>>> stuff = [2 , 'hello' , 2.718]
```

Notice that `stuff` is a variable that is assigned to be a list, and that list contains two numbers and a string. Python has no objection to different kinds of things being inside the same list! In fact, `stuff` could even have other lists in it, as in this example:

```
>>> stuff = [2 , 'hello' , 2.718 , [1 , 2 , 3]]
```

This ability to have multiple types of data living together in the same list is called *polymorphism* (meaning "many types"), which is a common feature in functional programming languages.

2.4.1 Some Good News!

Here's some good news: Almost everything that works on strings also works on lists. For example, we can ask for the length of a list just as we ask for the length of a string:

```
>>> len(stuff)
4
```

Notice that the list `stuff` really only has four things in it: The number `2`, the string `'hello'`, the number `2.718`, and the list `[1, 2, 3]`. Indexing and slicing also work on lists just as on strings:

```
>>> stuff [0]
2
>>> stuff [1]
'hello'
>>> stuff [2:4]
[2.718, [1, 2, 3]]
```

Just like strings, lists can be added and multiplied as well. Adding two lists together creates a new list that contains all of the elements in the first list followed by all of the elements in the second. This is called *list concatenation* and is similar to string concatenation.

```
>>> mylist = [1, 2, 3]
>>> mylist + [4, 5, 6]
[1, 2, 3, 4, 5, 6]
>>> mylist * 3
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

Finally, it's worth noting that doing something like:

```
>>> mylist + [4 , 5 , 6]
```

doesn't actually change `mylist`. Instead, it gives you a NEW list which is the result of concatenating `mylist` and the list `[4, 5, 6]`. You can verify this by typing `mylist` at the Python prompt and seeing that it wasn't changed:

```
>>> mylist + [4 , 5 , 6]
[1, 2, 3, 4, 5, 6]
```

```
>>> mylist
[1, 2, 3]
```

2.5 Functioning in Python

We've covered the basics of Python including how to represent and work with several types of data including numbers, strings and lists! Next, we're going to write actual programs. Throughout this and the remaining sections, we will keep in mind the problem that is motivating us in the first place—calculating edit distance—by looking at examples that help us build toward a solution to that problem. However, we wouldn't want you to get bored by looking at just one single problem, so we'll also throw in some other interesting ones along the way.

Let's begin with an analogy to something that you know and love: mathematical functions. In math, a function can be thought of as a "box" that takes some data as input, or argument, and returns some data as output, which we call its result, `return` value, or simply value. For example, the function $f(x) = 2x$ has an argument called x, and for any value that we "stick into" x we get back a value that is twice as large. Python lets us define functions too, and as in math, these functions take some data as arguments, process that data in some way, and then return some data as a result. Here's a Python function that we've named f, which takes an argument we are calling x and returns a result that is two times x.

In other words:

```
def f ( x ):
    return 2 * x
```

In Python, the syntax for functions uses the special word `def`, which means: "I'm `def` ining a function." Then comes the name that we've chosen for the function; in our case we chose to call it f. Then come the arguments to the function in parentheses (just like the definition of a function in math!), followed by a colon (":"). Then, we start a new line, indented a few spaces, and begin the function. In this case, our function computes 2*x and then returns it. The word `return` tells Python to give us back that value as the function's result. *Don't forget to indent the line (or lines) after the first line.* Python absolutely demands this. We'll talk a bit more about indentation shortly. We can now run the function in the Python interpreter like this:

Actually, we'll learn later that unlike in math, Python functions do not have to take arguments or return results. But for now we'll focus on functions that do.

```
>>> f (10)
20
>>> f ( -1)
-2
```

When we type `f(10)` we say that this is a *function call* because we are metaphorically placing a telephone call to f with argument 10 and asking it to give us an answer. When we call `f`, Python takes the value inside the parentheses and assigns it to the name `x`. Then it executes the statement or statements inside the function in order until there are no more statements to execute in the function. In this case, there is only a single statement inside the function, which doubles the value of `x` and returns the result to where the function was called.

The best way to define a function is to open up an editor and define the function there. If you are using IDLE go to the File menu and select "New Window." A new window will open up. Now you have two windows: the Python interpreter that you had originally and a new editor window where you can type your function definitions. In the editor window, you can edit comfortably, moving the cursor with the arrow keys and mouse. When you've completed your function, use the "File" menu to "Save" the file. Then, you can click on "Run" and the function will

The Python IDLE environment is named after Eric Idle, one of the members of the Monty Python comedy group.

be available for you to call in the original Python window. (If you are using some other version of Python, your professor will need to provide you with instructions on how to open a file for editing a function.)

As we noted, you can name a function more or less anything you want (as with variable names, there are a few exceptions, but it's not worth enumerating them). In this example, we called the function f simply to make the analogy with the mathematical function $f(x) = 2x$ that we defined at the beginning. To be honest, it's better to give functions descriptive names, just as we've advocated doing for variables. So, calling the function something like `double` would probably have been a better choice.

2.5.1 A Short Comment on Docstrings

A function is actually a computer program. We've just written our first program! Admittedly, a program that doubles a number is probably not one that you would show off to your friends and family. However, we're getting closer to writing some amazing programs. As our programs become more interesting, it will be important for their users to be able to quickly understand what the program is for. To that end, every function that we write from now on will begin with something called a *docstring*, which stands for "documentation string". Here's our snazzy doubling function `f` with its docstring:

```
def f ( x ):
    ''' Takes a number x as an argument and returns 2*x.'''
    return 2 * x
```

The docstring is a string that begins and ends with three single quote marks. If you now type `help(f)`, the docstring will be displayed. *You should always provide a docstring for every function that you write*.

2.5.2 An Equally Short Comment on Comments

Docstrings provide a way for users of your program to find out what the function is about. In addition to docstrings, you should always leave yourself (the programmer) or other programmers little notes, called *comments*, which explain the internal details of your program. Any text following a # is understood by Python to be a comment. The comment lasts until the end of that line. Python doesn't try to read or understand the comment (though it would be neat if it could!). Just to be clear, the difference between a docstring and a comment is that a docstring is intended for the user of the function. The user might

not even understand Python. A comment lets the programmer share details about how the program works with other people who might want to understand or modify it.

2.5.3 Functions Can Have More Than One Line

The function above, f, had only one statement, which doubled its argument and returned that value. However, in general functions are not limited to a single Python statement, but rather may have as many statements as you choose. For example, we could have written the exact same function from above as follows:

```
def f(x):
    '''Takes a number x as an argument and returns 2*x.'''
    twoTimesX = 2 * x
    return twoTimesX
```

Notice that when there is more than one statement in the function they must all be indented to the same position. This is how Python knows which statements are inside the function and which are not. We will say even more on indentation below. Here, you might be wondering which definition for f is better: the one with one line or the one with two. The answer is that both are equally valid and correct. Which you prefer is really up to you, the programmer. Some people prefer the one-line implementation because it's more compact, while others prefer to the two-line implementation because they prefer to store the values of intermediate calculations rather than immediately returning them. There's always more than one way to write a program!

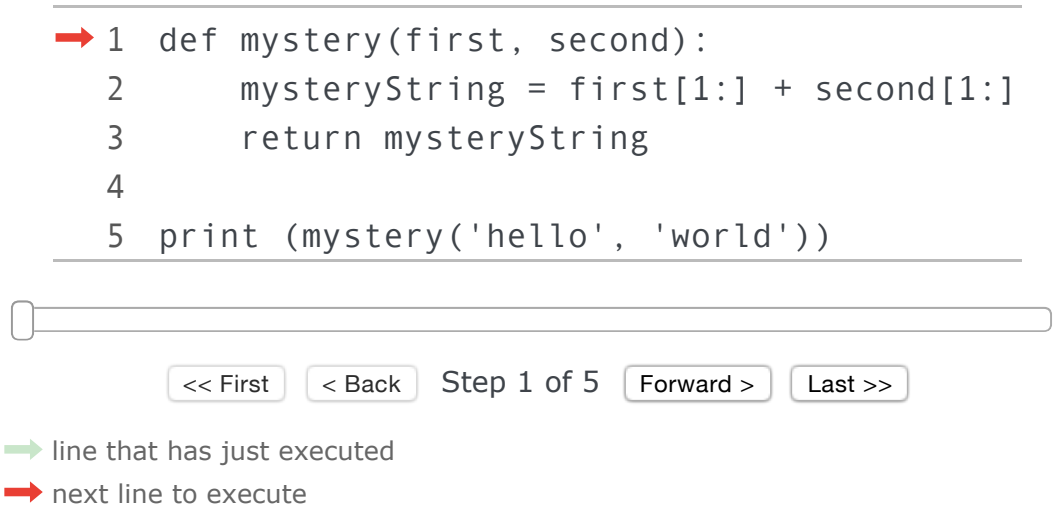
2.5.4 Functions Can Have Multiple Arguments

Just as in math, functions can have more than one argument. For example here's a function that takes two arguments, x and y, and returns $x^2 + y^2$:

```
def sumOfSquares (x , y):
    ''' Computes the sum of squares of its arguments '''
    return x**2 + y**2 # Here's our first comment!
```

In fact, the arguments to a function need not be numbers. They can be strings, lists, and a number of other things (more on this in the next chapter). Here's a mystery function. It takes two strings as arguments and returns another string. What is it doing? Once you think you have an idea, run the Codelens example and see if it matches your

expectations. Codelens will allow you to step through the program and visualize what the computer is doing at each step as it executes the program.



(ch02_mystery)

What's this `print` thing? In Python, `print` is a function that takes an argument (e.g., a number, a string, or any other value) and prints it on the screen. In this example, the argument that is being printed is whatever is being returned by the `mystery` function. There are two important things to note about `print`: First, it's a function, so its argument must be in parentheses. Second, it's different from `return` in an important way. The `return` statement is not a function - it simply leaves the function and returns a value to whoever called that function. In contrast, `print` is a function that displays a value on the screen. You can have many uses of `print` inside a function. Each one will `print` what it's told to print and then the function will continue on from there. On the other `return` is powerful stuff. The moment that a `return` statement is encountered, the function returns the value and it's done.

2.5.5 Why Write Functions?

At this point you might be wondering why we write functions. If we wanted to calculate twice the values of 10 and -1, wouldn't it be easier to just type what we have below rather than going through the hastle of defining a function?

```
>>> 2 * 10
20
```

```
>>> 2 * -1
-2
```

In this case, perhaps direct (or even mental!) calculation is simpler, but in general (and as we will shortly see), functions do much more complicated calculations that would be a pain to have to type over and over. Functions allow us to "package up" a bunch of calculations that we know we will want to perform over and over.

Remember our spiel about abstraction in Chapter 1? Packaging a computation into a function like this is a form of abstraction: we're hiding the details (the precise calculations in the body of the function) so that whoever calls the function can focus on its end result, instead of exactly how that result is calculated.

2.6 Making Decisions

So far our programs have just made straightforward calculations. But sometimes we need to write programs that make decisions, so that they perform different actions depending on some condition. For example, to solve our edit distance problem, we'll need to compare characters in our two strings and take different actions depending on whether those characters are the same or different. Before we get back to edit distance, let's consider a famous problem in mathematics called the "3n + 1" problem. It goes like this. Consider a function that takes a positive integer n as an argument. If the integer is even, the function returns the value n/2. If the integer is odd, the function returns 3n + 1. The problem, which is really a conjecture, states that if we start with any positive integer n and repeatedly apply this function, eventually we will produce the value 1. For example starting with n = 2, the function returns 1 because n is even. Starting with n = 3, the first application of the function returns 10. Now applying the function to 10 gives 5; applying it to 5 gives 16, which in turns gives 8, then 4, then 2, and finally 1. Ta-dah!

This seemingly benign little problem also has many other names, including the "Collatz problem", the "Syracuse problem", "Kakutani's problem", and "Ulam's problem". So far, nobody has been able to prove that this conjecture with many names is true in general. In fact, the famous mathematician Paul Erdos stated that "Mathematics is not yet ready for such problems."

It seems like too many big names for just one little problem!

Let's write the function in Python and then you can experiment with it!

```
1 def collatz( n ):
2     '''Takes a single number as argument and applies Collatz to i
3     if n % 2 == 0:
4         return n/2
5     else:
6         return 3*n + 1
7
8 # Change the code below to try it out...
9 print (collatz(3))
10
```

Run

My guess is that two equals signs means "very equal"

Before we look at this in detail, notice the expression `n % 2 == 0`. Recall that `n % 2` is just the remainder when n is divided by 2. If it is even, its remainder when divided by 2 will be 0; if it is odd, the remainder will be 1.

OK, but how about the == ? You'll recall that a single = sign is used in assignment statements to assign the value of an expression on the right-hand side of = to the variable on the left-hand side. The syntax == is doing something quite different. It evaluates the expressions on both sides of the == sign and determines whether or not they have the same value. This kind of expression always evaluates to either `True` or `False`. You might wish to pause here and try this in the Python interpreter. See what Python says to `42 % 2 == 0` or to `2 * 21 == 84/2` or to `42 % 2 == 41 % 2`. The special values `True` and `False` are called Boolean values. An expression that has a value of `True` or `False` is called a *Boolean expression.*

George Boole (1815-

By the way, the Booleans `True` or `False` should not be thought of as either numbers or strings. They are special kinds of values that

are intended for letting your function "reason" logically.

1864) was an English mathematician and philosopher. His work in logic forms part of the foundation of computer science and electrical engineering.

Back to our `collatz` function. It begins with an if statement. Right after the Python keyword `if` is the Boolean expression `n % 2 == 0`, followed by a colon. Python interprets this *conditional statement* as follows: "If the expression that you gave me (in this case, `n % 2 == 0`) is `True` then I will do all of the stuff that is indented on the following lines." In this case there is only one indented line, and that line says to return the value `n/2`. On the other hand, if the Boolean expression that was tested had the value *False*, Python would execute the indented lines that come after the else: line. You can think of that as the "otherwise" option. In this case, the function returns `3*n+1`.

It turns out that `else` statements are not required by Python, and sometimes we can do without them. For example, take a look at a slight modification of our collatz program shown below.

```
def collatz( n ):
    '''Takes a single number as argument and applies the Collatz to it'''
    if n % 2 == 0:
        return n/2
    return 3*n + 1
```

If `n` is even, this function computes `n/2` and returns that value; returning that value causes the function to end—which means that it stops evaluating any more statements. This last sentence is important and often confusing so let's highlight it again: **Executing a return statement always causes a function to end immediately**. Thus, if `n` is even, Python will never get to the line `return n/2`. However, if `n` is odd then the expression `n % 2` evaluates to `False`. In this case, Python drops down to the first line after the `if` statement that is at the same level of indentation as the `if`; in this case this is the line `return 3*n+1`. So we see that this version of the function behaves just like the first version with the else clause. But forty-one out of forty-two surveyed computer scientists advocate using the else version simply because it is easier to read and understand.

How do you humans know about 42? Has someone told you that it is the answer to the ultimate question of the universe, and everything?! I'll have to Google 42 to see what it says.

2.6.1 A second example

Now let's consider a second example, more closely related to the edit distance problem we are building up to. Our next function will determine whether the first character in each of two strings is the same or different. For example:

```
>>> matchFirst ( 'spam' , 'super')
True
>>> matchFirst ( 'AAGC' , 'GAG')
False
```

Here's one way to write the `matchFirst` function:

```
def matchFirst ( s1 , s2 ):
    '''Compare the first characters in s1 and s2
    and return True if they are the same.
    False if not'''

    if s1 [0] == s2 [0]:
        return True
    else :
        return False
```

As always, there's more than one way to write a program. Take a look at this shorter (and seemingly stranger) version:

```
def matchFirst ( s1 , s2 ):
    '''Compare the first characters in s1 and s2
    and return True if they are the same.
    False if not'''

    return s1[0] == s2[0]
```

What's going on here!? Remember that our goal is to return a Boolean—a value that is either `True` or `False`. The expression `s1[0] == s2[0]` is either `True` or `False`, and whichever it is, that's what we want to return. So the statement `return s1[0] == s2[0]` will produce our desired answer.

However, both versions of our `matchFirst` function have a subtle problem: there are some arguments on which the functions will fail to work. Take a moment to see if you can identify a case where things will go awry before you read on.

Did you find the problem? When either (or both) of our arguments are the empty string (a string with no characters inside it), Python will complain. Why? The empty string has no symbol at index 0—because it has no symbols in it at all! (Remember, the symbol at index 0 actually refers the first symbol in the string. When we index into strings, we are actually measuring the distance from the beginning to the desired character, and the first symbol is at a distance of zero characters from the start.)

Many computer scientists confuse measuring with counting and also count from zero!

We can fix this by using another if statement to check for the special case that one or both of the arguments is the empty string. The built-in `len` function will tell us whether the string has length 0. It's a bit weird, but a string of length 0 has no symbols in it at all, not even one at index 0.

```
def matchFirst ( s1 , s2 ):
    '''Compare the first characters in s1 and s2
    and return True if they are the same.
    False if not'''

    if len(s1) == 0 or len(s2) == 0:
        return False
    else :
        return s1[0] == s2[0]
```

Now, if either string is empty the function will return `False`. Notice the use of the word `or` in the `if` statement here. It's saying "if the length of s1 is 0 or the length of s2 is 0 then return *False*."

if **both** *strings are empty, the function returns* `false`. *Do you think this is the correct result? How would you change it to return* `True` in that case?

Python expects that the condition being tested in an `if` will have a Boolean value—that is, its value is either `True` or `False`. In this case, `len(s1) == 0` has a Boolean value; it's either `True` or `False`. Similarly, `len(s2) == 0` is Boolean. The connector or is Boolean "glue" much like addition is arithmetic glue. Just as the plus sign adds the two numbers on its left and right and gives us back another number (the sum), the or sign looks at the Booleans on its left and right and gives us back another Boolean— `True` if at least one of the Booleans is `True` and otherwise it gives us back `False`.

By the way, Python also has another piece of Boolean glue called `and`. Not surprisingly, it gives us back `True` if both of the Booleans on its left and right are `True` and gives us back

`False` otherwise. So, the statement:

```
len(s1) == 0 and len(s2) == 0
```

will be `True` if *both* strings are empty.

Finally, Python has something called not that "negates" a Boolean, flipping `True` to `False` and `False` to `True`. For example:

```
not 1 == 2
```

will be `True` because `1 == 2` is `False` and its negation is therefore `True`. Incidentally, we can mix and match our new friends `or`, `and`, and `not`. For example, the expression:

```
1 == 2 or not 41 == 42
```

will evaluate to `True` because although `1 == 2` is `False`, not `41 == 42` is `True`, and the result of `or` -ing `False` and `True` is `True`.

But later we'll see a better way to write `not 41 == 42`

2.6.2 Indentation

We have noted that Python is persnickety about indentation. After the first line (the one containing the `def`), Python expects all of the remainder of the function to be indented. As we've seen above, indentation is also used in `if` and `else` statements. Immediately after an `if` statement, we indent the line or lines that we want Python to execute if the expression is `True`. Similarly, the line or lines that should be executed after the *else* are indented as well.

For example, here is another way that we could have written our `collatz` function; this one uses one line to compute the desired value and then returns it in a second. Most programmers wouldn't write the function this way because it's more verbose than necessary, but when programs get more involved and complicated it can be handy (and sometimes necessary) to have multiple lines like this. And again, if you prefer to write it this way even in this simple case, there's nothing wrong with that!

```
def collatz(n):
```

```
    '''Takes a single number as an argument and
       applies the Collatz function to it.'''
    if x % 2 == 0:
        result = 3*n + 1 # Create a variable called result
        return result    # Now we return the value of result
    else:
        result = n/2     # Create a variable called result
        return result    # Now we return the value of result
```

2.6.3 Multiple Conditions

We noted that sometimes we use `if` without a matching `else`. On the flip side, sometimes it's useful to have more than one alternative to the condition tested in the `if` statement.

Consider again our edit distance problem. While we're not quite ready to solve the whole thing yet, we can solve it for the case that one or both of the argument strings is empty. In this case, the edit distance between the two strings is simply the length of the non-empty one, because it necessarily will take that many insertions to the empty string (or, conversely, deletions from the non-empty one) to make the two the same.

Here is a function that solves the edit distance problem only in this simple case:

```
def simpleDistance(s1, s2):
    '''Takes two strings as arguments and returns the edit
       distance between them if one of them is empty.
       Otherwise it returns an error string.'''
    if len(s1) == 0:
        return len(s2)
    elif len(s2) == 0:
        return len(s1)
    else:
        return 'Help! We don't know what to do here!'
```

Let's dissect this function.

Dissection!? Is this CS or biology?

It starts by checking whether the string `s1` is empty; if so the function returns the length of `s2`. If `s1` is not empty, then the function will use an *elif* statement to see whether `s2` is empty. If so, it will return the length of `s1`. Finally, if neither `s1` nor `s2` is empty, it will return a string reporting that it does not know what to do.

(By the way, notice that if *both* strings are empty then we'll return 0, which is the correct answer. Do you see why 0 gets returned?)

Computer scientists are infamous for claiming that anything that their software does is a "feature"—making even their mistakes sound like they were intentional and useful!

The `elif` statements are pronounced "else if". The `elif` in this function is only reached if `s1` was not empty. The `elif` is saying "else (otherwise), if the string s2 is empty, then return the length of s1." Although we only have one `elif` in this function, in general, after an `if` we can have as many `elif` conditions as we wish. Then, at the end we can have zero or one `else` statements, but not more than one! The final `else` statement specifies what to do if all of the preceding conditions failed.

There's nothing wrong with a function returning a string! Returning means we're done—the function is over and we get back the value in the *return* statement. Generally you would not want your function to return a string in some cases and a number in others (like ours currently does), but in this case we're doing that just to make it clear what's happening in each case. Our final edit distance function won't have this "feature."

A Colorful Application of if, elif, and else

In 1852, Augustus DeMorgan revealed in a letter to fellow British mathematician William Hamilton how he'd been stumped by one of his student's questions:

> *A student of mine asked me today to give him a reason for a fact which I did not know was a fact—and do not yet. He says that if a figure be anyhow divided and the compartments differently coloured so that figures with any portion of common boundary line are differently coloured—four colours may be wanted, but not more...*

DeMorgan had articulated the *four-color problem*: whether or not a flat map of regions would ever need more than four colors to ensure that neighboring regions had different colors. The problem haunted DeMorgan for the rest of his life, and he died before Alfred Kempe published a proof that four colors suffice in 1879.

Kempe received great acclaim for his proof. He was elected a Fellow of the Royal Society and served as its treasurer for many years; he was knighted for his accomplishments. He also continued to investigate the four color theorem,

publishing improved versions of his proof and inspiring other mathematicians to do so.

However, in 1890 a colleague showed that Kempe's proof (and its variants) were all incorrect—and the mathematical community resumed its efforts. The four-color problem stubbornly resisted proof until 1976, when it became the first major mathematical theorem proved using a computer. Kenneth Appel and Wolfgang Haken of the University of Illinois first established that *every flat map* must contain one of 1,936 particular sub-maps that they defined. They next showed, using over 1200 hours of computer time, that each of those 1,936 cases could not be part of a counterexample to the theorem. Four colors did, in fact, suffice!

Their program essentially used a giant conditional statement with 1,936 occurrences of *if*, *elif*, or *else* statements. Such *case analyses* are quite common in computational problems (although 1,936 cases is admittedly more than we might usually encounter).

We suspect that DeMorgan would feel better knowing that the question he couldn't answer wouldn't be answered at all for over a century.

2.7 Recursion!

So far we've built up some powerful programming tools that have allowed us to do some interesting things, but we still can't solve the edit distance problem except for the very special case that one of the strings is empty.

What about the cases we really care about, where neither string is empty? Imagine for a moment that we know that both of our strings are exactly four characters long. In that case, we don't need any deletions or insertions to get from the first string to the second string—we only need substitutions. For example, to get from "spam" to "spim" we just substitute the "a" with an "i". For the case of two strings of length four, we could compute the distance this way:

```
def distance(s1, s2):
    '''Return the distance between two strings,
       each of which is of length four.'''
    editDist = 0
    if s1[0] != s2[0]:
```

```
        editDist = editDist + 1
    if s1[1] != s2[1]:
        editDist = editDist + 1
    if s1[2] != s2[2]:
        editDist = editDist + 1
    if s1[3] == s2[3]:
        editDist = editDist + 1
    return editDist
```

The notation `!=`, which is read as "not equals", offers a much nicer notation than the clumsy but equivalent `not s1[0] == s2[0]`. Notice that this function starts by setting a variable named `editDist` to 0 and then adds 1 to that variable each time it finds a pair of corresponding symbols that don't match and thus require a substitution. (By the way, also notice that we used *if* s and not `elif` s. Do you see why `elif` s would give us the wrong answer here?)

While this program will work for four-letter words, it doesn't help us if the words are longer or shorter. Moreover, it can't deal with insertions or deletions. We might be tempted to add more `if` s to handle longer strings, but how many would we add? No matter how many we added, we would still run into trouble for sufficiently long strings.

To add both the ability to handle strings of arbitrary length and the ability to handle insertions and deletions, we will need a beautiful and elegant new ingredient called *recursion*.

Before using recursion for the edit distance problem, let's take an excursion to visit a group of aliens who have come to Earth from a distant planet for the debut of the seventeenth Harry Potter movie.

A recursion excursion!

At the moment there are, let's see, 42 aliens in line. One alien muses to itself, "I wonder how many different ways I could arrange us 42 aliens?"

You may know the answer: it's $42 \times 41 \times 40 ... 3 \times 2 \times 1$, also known as "42 factorial" and written $42!$. (There are 42 choices for the alien that could be first in line, 41 who could get the next spot, and so forth.)

The alien decides that it would like to write a Python program to compute the factorial of any positive integer. Fortunately, the alien has done some shopping at the local mall, where it purchased a laptop that runs Python. Unfortunately, the alien is vexed by how to write such a program. Fortunately, we observe that $42! = 42 \times 41!$ and in general, $n! = n(n-1)!$. That is, the factorial of n can be expressed as n times the result of

solving another smaller factorial problem. This is called a *recursive definition*: it expresses the problem in terms of a smaller version of the same problem; the definition *recurs* in the solution.

Before writing a program to compute the factorial, let's just observe that this recursive definition is indeed useful. Imagine that we want to compute $3!$. According to the recursive definition, that's just $3 \times 2!$. So now we're off to find $2!$. Using the same definition again, we see that $2! = 2 \times 1!$. If we can figure out what $1!$ is, we'll be in good shape. According to the definition, $1!$ is $1 \times 0!$. According to the definition, $0!$ is $0 \times (-1)! \ldots$ Uh oh, this is bad—the process will never stop. Moreover, we are not really interested in $0!$ or the factorial of a negative number.

Unfortunately, it's not clear to me how this helps.

To work around the difficulty, we should add a rule that tells us when to stop this recursive process. A reasonable stopping place is to say, "If you get to the point that you're trying to compute `1!`, stop using the recursive rule and just report that the answer is 1." This is called the *base case* of the recursive definition. It tells us when to stop applying the rule. Of course, we should always check the base case *before* deciding whether to continue applying the recursive rule.

Coming back to the example of $3!$, we get down to the case of $1!$ and the base case says "Aha! Stop! That's 1." So, we have $1! = 1$. Remember that it was $2! = 2 \times 1!$ that wanted to know the value of $1!$. So now we plug the 1 in for $1!$ and determine that $2! = 2 \times 1 = 2$. But $3! = 3 \times 2!$ was the one that asked about $2!$ and is waiting patiently for the answer. So, now $3!$ determines that its result is $3 \times 2 = 6$. That's it, we've computed $3!$ using the recursive definition.

Let's try to capture the recursive definition as closely as we can in Python. This may seem weird or even downright wrong at first, but let's try. Here's the program. (Run it, and try it out!)

```
1  def factorial(n):
2      '''Recursive function for computing
3      the factorial of n.'''
4
5      if n == 1:
6          return 1
7      else:
8          result = n * factorial(n-1)
9          return result
10
11 print (factorial(4))
```

<< First < Back Step 1 of 17 Forward > Last >>

line that has just executed

next line to execute

Frames Objects

(ch02_secondexample)

Looking at this function, we see that it looks like a translation from math into Python. Try running this function. The factorial of 5 is 120 and the factorial of 70 is larger than the number of particles in the universe—but Python will gladly compute it.

The fact that this function correctly computes the factorial function might seem mysterious and magical. We will see shortly that there is no magic here and not even any mystery! However, for just a moment, let's take a leap of faith that Python will do what we intend when we run this function (hopefully corroborated by your computational experiment that the function seems to work on the arguments that you tried). In a moment we'll come back to convince ourselves that this recursion really *must* work.

A typical recursive function has two main parts:

- **A base case** : This is the value that the function returns for the "simplest" argument.
- **A recursive step** : This is the solution to a smaller version of the problem, computed by calling the function with a smaller or simpler argument. This solution to the smaller problem is then used in some way

to solve the original problem.

In the factorial function, the base case is when `n` is 1. In this case, our function simply returns 1, since $1! = 1$, and we're done. The recursive step, the part inside the `else` statement, computes the factorial of `n-1`, multiplies the result by `n`, and returns this value.

Now let's return to the edit distance function. We're still not ready to solve it in its entirety, but we're getting closer! Let's now consider the situation where the two strings are guaranteed to be of the same length (so that no insertions or deletions need to be used), but their length could be anything—not just four! In this case, the edit distance is the number of positions where the two strings differ.

The base case now is when both strings are empty—that's the simplest case in which the two strings could have the same length. In that case, the edit distance is 0 and we're done.

If the base case does not apply—that is the strings are of some non-zero length—then what happens next depends on whether or not the two strings match at position 0. If they don't match, that position contributes 1 to the edit distance. Now, we have to compare the remainder of the two strings to one another to find the number of differences between them. But that's exactly the same problem, just for the two strings with their leading symbols chopped off! So, if the characters at position 0 don't match, the edit distance between `s1` and `s2` is 1 plus the edit distance between `s1[1:]` and `s2[1:]`. Remember, the notation `s1[1:]` is a string just like `s1` except with the symbols at position 0 chopped off.

On the other hand, if the two strings match on the first symbol then the edit distance between `s1` and `s2` is just the edit distance between `s1[1:]` and `s2[1:]`. This results in the function below.

```
def simpleDistance(s1, s2):
    '''Takes two strings of the same length and returns the
       number of positions in which they differ.'''
    if len(s1) == 0:      # len(s2) is also 0 since strings
                          # have the same length
        return 0          # base case
    elif s1[0] != s2[0]: # recursive step, case 1
        return 1 + simpleDistance(s1[1:], s2[1:])
    else:                 # recursive step, case 2:
                          #   s1[0] == s2[0]
        return simpleDistance(s1[1:], s2[1:])
```

Takeaway message : *The secret to thinking about recursion is to ask yourself "would it help if I had the answer to a slightly smaller version of the same problem?" If the answer is "yes," then you can write a recursive function that calls itself to get the answer to the slightly smaller version of the problem and then use that result to solve your original problem.*

And don't forget the base case!

2.8 Recursion, Revealed

2.8.1 Functions that Call Functions

We promise that in this section we will reveal the "magic" behind recursion. But for right now, we've consulted with our lawyers and they told us that first we could sneak in a short section on a slight tangent. Actually, it's not quite as much of a tangent as it may seem at first. Did you know that the word tangent was evidently first used in 1583 by the Danish mathematician Thomas Fincke? Speaking of Denmark, did you know that Legos were invented there? We digress.

Take a look at the code in the example below. How did Python arrive at 42 when we called `demo` with argument 13? Does Python get upset or confused by the fact that each of the functions here has a variable named *x* and a variable named *r*? How does changing the value of *x* in one function affect the value of *x* in another function?

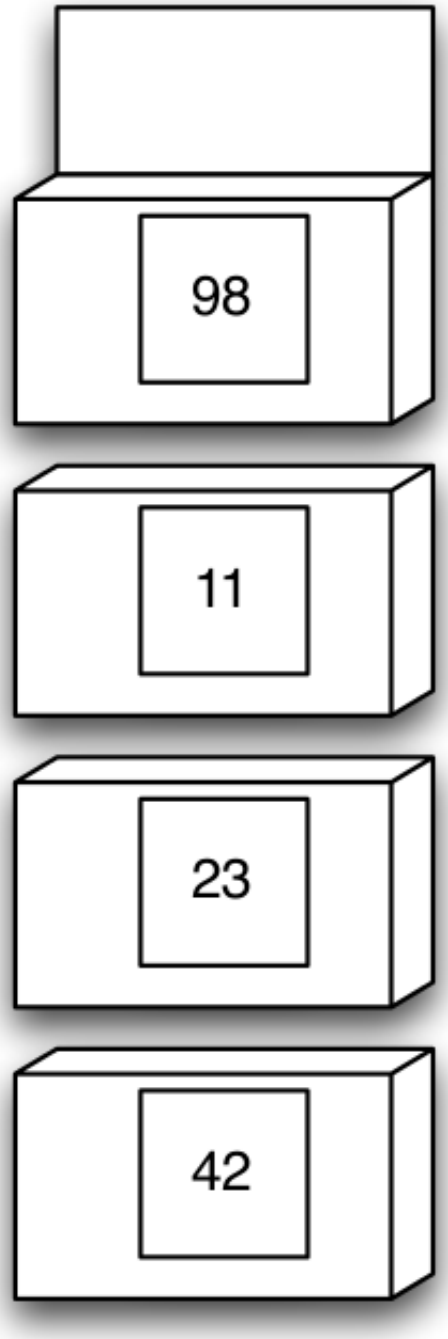


Figure 2.1: A stack of storage boxes. Only the box on the top has its door accessible.

```
def demo(x):
    r = f(x+6) + x
    return r

def f(x):
    r = g(x-1)
    x = 1
    return r + x

def g(x):
    r = x + 10
```

```
    return r
```

```
>>> demo(13)
42
```

The secret here has to do with the way that Python (and, indeed, any self-respecting programming language) deals with variables. Python has a large supply of metaphorical storage boxes that it can use to store "precious" commodities.

These storage boxes have the special feature that their doors are on top and they are stackable. Figure 2.1 shows an artist's rendition of a stack of boxes. The box on the top of the stack is the only one that you can tinker with. You'll have to remove that box before you can tinker with the contents of the one below it. However, we've put little windows in all of the boxes just for the sake of explanation—allowing you to peek at what's in there.

And letting the numbers get some natural light!

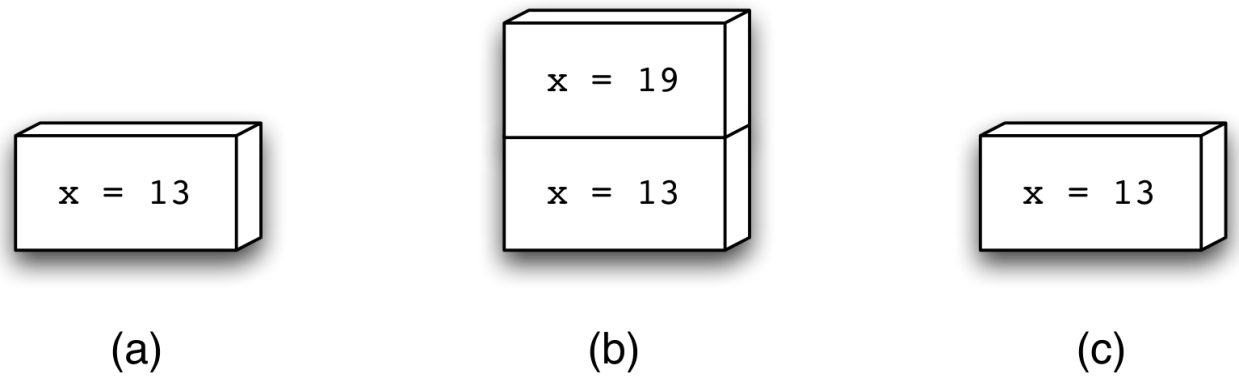


Figure 2.2: The use of stack when `demo(13)` is evaluated. (a) `demo` places its value of `x` on the stack. (b) `f` places its value of `x` in a box on top of the stack. (c) The stack after `g` is done and `f` has retrieved its value of `x`.

Storage boxes? Doors? Windows? Are you CS professors crazy? Perhaps, but that's not really relevant here. Let's take a closer look at our program above. When the call `demo(13)` is made, the value 13 is passed into the `demo` function's argument named `x`. This variable is owned by `demo` and cannot be seen by "anyone" outside of this function.

Python, like most programming languages, likes to enforce privacy.

The function `demo` needs to call `f(x+6)` since this is the first expression in the right-hand side of the statement `r = f(x+6) + x`. However, `demo` wants to ensure that the "precious" value of its variable *x*, currently, 13, is preserved. It rightfully worries that it might get changed by the function `f` or perhaps one of `f`'s pernicious friends (like *g*). So, right before the function call to `f`, Python automatically stores all of `demo`'s variables in a storage box for safe-keeping. In this case, there is a variable called `x` and Python stores its value, 13. This is shown in Figure 2.2(a).

When the function `f` is called, it gets the argument $13 + 6 = 19$. The value 19 is going into *f* 's argument *x*. Now *x* has the value 19. The function `demo` is not worried about this change in the value of *x* because it has locked up its own value of *x* in its secure box. It will retrieve that value when *f* is done and returns control to `demo`.

Next, `f` calls `g(19-1)`. Again, before doing so, it saves its own value of *x*, 19, in its own box for safe-keeping. This box is stacked on top of the previous box as shown in Figure 2.2(b). Now function `g` gets 18 in its argument of *x*. It computes $10 + 18 = 28$ and returns that value. When we return to function *f*, that function immediately finds the storage box on the top of the stack, retrieves its value of *x* from the box, and then removes that box from the stack. Now, *f* has restored its original value 19 for *x*. The current situation is shown in Figure 2.2(c). Now *x* is changed to 1—though that's kind of silly since it's about to be thrown away. Finally *f* returns $28 + 1 = 29$ to the `demo` function. At this point *demo* finds its box at the top of the stack, opens it, restores its original value of *x* to 13, and tosses the box. This value is now used in the rest of computation, and the *demo* returns the value $29 + 13 = 42$. Whew!

In computer science, this pile of storage boxes is called the *stack*. In practice it is implemented using the computer's memory rather than storage boxes with cute doors and windows, but we like the metaphor.

Where a variable's value can be seen is called its *scope*. Our example demonstrates that the scope of a variable is limited to the function in which it resides. That's all cool, but our lawyers have warned us that if we don't talk about recursion now, you'll have grounds to sue us, so here we go!

2.8.2 Recursion, Revealed, Really!

OK, now back to our first recursive function, factorial:

```
1  def factorial(n):
2      '''Recursive function for computing the fa
3
4      if n == 1:
5          return 1
6      else:
7          result = n * factorial(n-1)
8          return result
9
10 print (factorial(4))
```

<< First | < Back | Step 1 of 17 | Forward > | Last >>

line that has just executed
next line to execute

Frames Objects

(ch02_factorial)

Amazing! But even though it *seems* to run correctly and implements a recursive definition that *seems* correct, the function still *seems* rather like magic.

So let's take a look at what Python is doing when we run `factorial(3)`. We'll explain it step-by-step and summarize the process in Figure 2.3.

To me it still seems unseemly!

The function begins with *n* equal to 3. Since *n* is not equal to 1, the condition in the `if` statement is `False` and we continue down to the `else` part. At this point, we see that we need to evaluate `n * factorial(n-1)` which requires calling the function `factorial` with argument 2. Python doesn't realize—or even care—that the function that is about to be called is the very same function that we're currently running. It simply uses the same policy that we've seen before: "Aha! A function call is coming up. I'd better put my precious belongings in a storage box on the stack for safekeeping so that I can retrieve them when this function call returns."

In this case, `n`, with value 3, is the only variable that `factorial` owns at the moment. So, `factorial(3)` puts the value of *n* equal to 3 away in the box for later retrieval. Then, it

calls `factorial(2)`. Now `factorial(2)` runs. That means that `factorial` starts at the beginning with an input of 2 so that *n* now has the value 2. Fortunately, the original value of *n* has been locked away for safekeeping because we'll need it later. For now, though, `factorial(2)` again goes to the `else` part where it sees that it needs to call `factorial(1)`. Before doing so, `factorial(2)` puts its value of *n*, namely 2, in its own storage box at the top of the stack. Then it calls `factorial(1)`.

Finally, `factorial(1)` is executed. Notice that *n* is now 1, so the expression `n == 1` evaluates to `True` and `factorial(1)` simply returns 1. But this 1 must be returned to the function that called it. That's true anytime there is a function call! Recall that `factorial(2)` made that call. At this point, control is returned to `factorial(2)`, which immediately goes to its storage box at the top of the stack, opens it, retrieves its value of *n* (which is 2), and discards the box from the top of the stack. Now, `factorial(2)` resumes its work. It computes *2 * 1*, assigns that value (2) to the variable `result`, and returns that.

That value is returned to the place where `factorial(2)` was called. That was in `factorial(3)`, which now goes to the top of the stack, opens the storage box, retrieves its value of *n*, and tosses the box. Now, *n* is 3 and `factorial(3)` computes *3*2*, which is 6, and returns that value. The value is returned to us because we called `factorial(3)` at the prompt and, voila, we have our answer!

Remember that the test `if n == 1`, the base case, is critically important.

Without it, the program has no way of knowing when to stop. In general, **we advocate trying to write the base case first** because it gets an easy case out of the way and lets you concentrate on the recursive part. Moreover, the base case needs to be the first thing that your recursive function tests, to make sure that it can eventually stop.

The base case in a recursive function is analogous to the base case in a proof by induction. In fact, you may have noticed that recursion and induction are very similar in spirit.

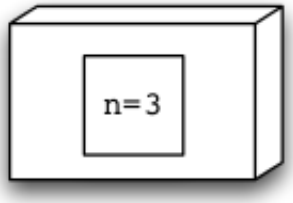
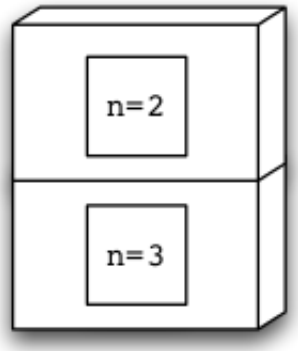
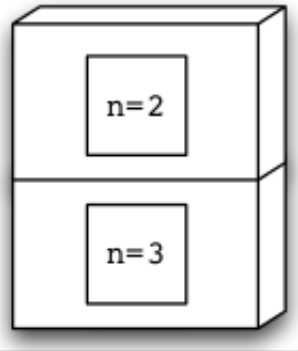
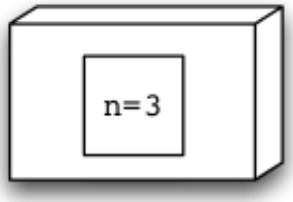
Function call	What happens here...	What the stack looks like at this mom
factorial(3):	Places its value of n=3 on the top of the stack and then calls factorial(2)	
factorial(2):	Places its value of n=2 on the top of the stack and then calls factorial(1)	
factorial(1):	Returns the value 1 to the place where it was called, namely factorial(2)	
factorial(2):	Takes the value 1 returned from factorial(1), finds its own value of n=2 at the top of the stack, computes the product, 2*1, and returns this value to the place where it was called, namely factorial(3)	
factorial(3):	Takes the value 2 returned from factorial(2), finds its own value of n=3 at the top of the stack, computes the product (3*2), and returns this value, 6, to the place where it was called...	

Figure 2.3: What happens when we invoke `factorial(3)`

One way to help you think of what the base case should be is to ask yourself the question, “What is the ‘easiest’ input that I might get for this problem?” In the case of `factorial`, it seems that $1!$ is the easiest reasonable factorial problem you can imagine.

(Although we could have also defined $0! = 1$ as the base case and this would have worked too. Indeed, a mathematician could give us convincing arguments why we should allow for $0!$.)

Takeaway message: *Recursion is not magic! It is simply one function calling another, but it just happens that the function that we're calling has the same name as the function that we're in!*

2.9 Building Recursion Muscles

Let's do another example with recursion to flex our recursion muscles. Imagine that we want to write a function that takes a string as an argument and returns the reversal of that string—that is, the string in reverse order. So, if we give this function "spam", it should return the string "maps." If we give it "alien", it should return "neila".

Did you know that "Neila" is a town in Spain? Population 235.

The secret to writing a recursive function is to try to identify how solving a "smaller" version of the problem would allow us to get closer to solving the original problem. In the case of computing the factorial, we observed that computing the factorial of n is easy if we know the factorial of $n - 1$. Aha! Now we can use recursion to compute the factorial of $n - 1$ and when we get that answer, we can multiply it by n and we've solved our problem. This is what is sometimes called the *recursive substructure property*. That's a fancy term that really means that computing the factorial of a number can be viewed as first computing a slightly simpler factorial problem and then doing just a bit of extra work (in this case, multiplying by n at the end).

Similarly, we'd like to find the recursive substructure in reversing a string. If we want to reverse a string like "spam", we observe that if we chopped off the first letter, resulting in "pam" and then reversed "pam", we'd be nearly done. The reversal of "pam" is "map". Once we have "map", we can add that chopped-off "s" to the *end* and we get "maps". Generalizing this, we can say that to reverse a string, we can chop off the first letter, reverse the remainder, and then add that first letter to the end.

Using this rule, we see that to reverse "spam" we will first reverse "pam" (and then add the "s" to the end). To reverse "pam", we'll first reverse "am" (and then add the "p" to the end). To reverse "am", we'll first reverse "m" (and then add "a" to the end). To reverse "m" what will we do? Well, continuing in this spirit, we'll chop the "m" off and we'll be left with ""—a string with no symbols. This is called the *empty string*. It seems like a strange and

perhaps even invalid string, but one could say the same thing about the number zero and yet we all agree that zero is a perfectly OK number.

But now we have a problem. How do we reverse the empty string? Simple: this is the base case of our recursion. If we're asked to give the reversal of the empty string, it's clearly just the empty string itself!

You Earthlings are geniuses for inventing zero. It's like nothing I've ever seen!

So, continuing with our example, when we reverse "", we return "". Now, we concatenate the "m" to that and we return "m". That's the reversal of "m". Recall that it was the reversal of "am" that requested the reversal of "m". Now that we have that, the reversal of "am" is "m" with the "a" concatenated at the end, which is "ma". It was the reversal of "pam" that was waiting patiently for the reversal of "am". "Thank you," it says. "I've been waiting for the answer, which I see now is 'ma' and I will concatenate my 'p' to the end of it and get 'map'." Finally, it was the reversal of "spam" that requested the reversal of "pam". It gets "map", concatenates the "s", and returns "maps". Wow!

While it's instructive to trace through this logic step by step to understand what's happening, it can be aggravating to do this every time you write a recursive function. So now we'll take a small leap of faith and try to write this recursive function in Python, based on our observation of the recursive substructure.

We've agreed that if the string is empty, the problem is easy—we just return the empty string and we're done. That's the base case and we take care of that first. If the string is not empty, we'll find the first symbol in the string and "store" it away for later. In other words, we'll have a variable that keeps that first symbol for later use. Then, we'll slice off the first symbol, reverse the remaining string, and add the first symbol to the end of the resulting string.

Remember that a string can be defined using either single quotes or double quotes. We'll use single quotes throughout, just for consistency.

```
def reverse(string):
    '''Takes a string as an argument and returns
       its reversal.'''
    if string == '':            # Is the string empty?
        return ''               # If so, reversing it is easy!
    else:
        firstSymbol = string[0] # Hold on to the first symbol
```

```
        return reverse(string[1:]) + firstSymbol
```

At this point you might be starting to notice a pattern for writing recursive functions with strings. The base case is (often) when the string(s) are empty, and the recursive call is (often) made on the string without its first character. We'll see this pattern come up again and again. What differs from function to function is what happens with the result of the recursive call, and what is returned in each case.

2.10 Use It Or Lose It

We're almost ready to completely solve our edit distance problem. In fact, we have all the programming tools we need to do so. However, the general edit distance problem is a substantially more complicated problem than reversing a string, and we're missing some problem-solving tools that will help considerably. So in this section we're going to introduce a general approach to solving a large class of problems (including the edit distance problem), which we call "Use It Or Lose It."

Our alien has a predicament: it's planning to return to its home planet soon (after the Harry Potter 17 debut and a massive shopping spree at the local mall) and has acquired way more stuff than will fit in its suitcase. The alien would therefore like to select a subset of items whose total weight is as close as possible to the suitcase capacity, but without exceeding it. (Greedy creature!)

For example, imagine that the suitcase capacity is 42 units and there are items with weights 5, 10, 18, 23, 30, and 45. In this case, the best we can do is to choose the weights 18 and 23 for a total of 41. On the other hand, if an item with weight 2 is also available, we can get exactly 42 by choosing the weights 2, 10, and 30.

So, what we'd like is a function that takes two arguments: a number representing the suitcase capacity and a *list* of *positive* numbers (in no particular order) representing the weights of the items. The function should then return the largest total weight of items that could be chosen without exceeding the suitcase capacity. We'll call our function *subset* and we can imagine using it (once it's written!) this way:

```
>>> subset(42, [5, 10, 18, 23, 30, 45])
41
>>> subset(42, [2, 5, 10, 18, 23, 30, 45])
42
```

A slightly more ambitious task would be to actually report the set of items that gives us this best solution, but let's not worry about that for now.

We start by thinking about the base case. What are the "easy" cases where the *subset* function needs to do almost no work? Clearly, if the capacity that we're given is 0, we can't take any items, so we should return 0 to indicate "sorry, the maximum sum that you can attain is 0." So, we can start this way:

```
def subset(capacity, items):
    '''Given a suitcase capacity and a list of items
       consisting of positive numbers, returns a number
       indicating the largest sum that can be made from a
       subset of the items without exceeding the capacity.'''

    if capacity == 0:
        return 0
```

Actually, we could also return zero if the capacity is *less than* zero; we'll see that below. But there's another "easy" case that we haven't handled. What if the list of items is empty? In that case, we also can't take any items. We could handle this with an *elif* after the *if* statement:

```
elif items == []:
    return 0
```

Alternatively, since we plan to return 0 if the capacity is less than or equal to 0 *or* the list is empty, we could simply modify the *if* statement above to say:

```
if capacity <= 0 or items == []:
    return 0
```

Recall that we advocate taking care of the base cases first, since this takes care of the "easy" situations. Notice that since there were two arguments to *subset*, we should expect that there will be two base cases to handle: either of the arguments could be "easy" (capacity is ≤ 0 or list of items is empty). Often, the number of base cases is equal to the number of arguments to the function.

OK, now for the actual recursion! Somehow we need to find the recursive substructure in this problem. Let's take a look at the first item in the given list. That list isn't necessarily in any particular order, but it's easy to pick on the first item so let's start with it. (We could

have just as well looked at the last item or any other, but Python makes it particularly easy to look at the first one, since it's just `items[0]`.)

If that first item happens to be larger than the capacity of the suitcase, we have no choice but to toss it out. In that case, we're confronted with a simpler problem: Find the best solution with the given capacity but with a list that has had the first item removed. We'd like to call some function to help us find that solution. What function can do that? Our own `subset` function! So, here's what we have so far:

```
def subset(capacity, items):
    '''Given a suitcase capacity and a list of items
       consisting of positive numbers, returns a number
       indicating the largest sum that can be made from a
       subset of the items without exceeding the capacity.'''

    if capacity <= 0 or items == []:
            return 0
    elif items[0] > capacity:
            return subset(capacity, items[1:])
```

Remember, there is no magic here! We're simply going to call a function that happens to be the same function as the one we are in.

Finally, if that first item, `items[0]`, is not greater than the capacity, we might want to use it, but not for sure. For example, if the capacity was 10 and the list of items was `[8, 4, 6]` we *could* use the item with value 8, but if we do use it we can't take any other items (because the remaining items in the list will exceed the remaining capacity). A better solution would be *not* to use it and take the items with values 4 and 6 to get a solution with total value 10. So, the question here is should we "use it or lose it"? (The "it" here being the item with value 8.)

We don't yet know the answer to that question. However, we can make *two* recursive calls, one that finds the best solution that "uses" the item with value 8 and one that finds the best solution that "loses" the value 8. The better of these two solutions must be the best solution overall. After all, with respect to that item with value 8, any solution must either use it or lose it!

The case that we lose the first item in the list is easy. We just want to find the best solution with the same capacity and the list `items[1:]`. If we choose to use that item, we now get the value of that item in our solution, but our capacity is now reduced by the

In this case, the

weight of that item. So our complete function will look like this:

recursion is trying to "e-value-8" the two options!

```
 1  def subset(capacity, items):
 2      if capacity <= 0 or items == []:
 3          return 0
 4      elif items[0] > capacity:
 5          return subset(capacity, items[1:])
 6      else:
 7          loseIt = subset(capacity, items[1:])
 8          useIt = items[0] + subset(capacity - i
 9          return max(loseIt, useIt)
10
11  print (subset(4, [1, 2]))
```

<< First | < Back | Step 1 of 32 | Forward > | Last >>

line that has just executed
next line to execute

Frames Objects

(ch02_using)

By the way, the function `max` is built in to Python; it can take any number of arguments and returns the maximum of all of them. In this case, we're using `max` to return the better of our two options: "use it" and "lose it."

The "use it or lose it" paradigm is a powerful problem-solving strategy. It allows us to exploit recursion to explore all possible ways to construct a solution. We're now (finally) ready to return to the problem that started off this chapter.

2.11 Edit distance!

Now that we've seen recursion and the "use it or lose it" strategy, we are ready to solve our original problem: Finding the edit distance between two strings. Recall that the objective is to find the least number of substitutions, insertions, and deletions required to

get from one string to another. As we noted at the beginning of the chapter, it doesn't matter whether we choose to change the first string into the second or vice versa, so we'll choose to start with the first string and try to transform it into the second string.

Just as a quick reminder, here's an example of the full version of the problem: Consider transforming the string "alien" into the string "sales." We can begin by inserting an "s" at the front of "alien" to make "salien". Then we delete the "i" to make "salen." Then we replace the "n" with an "s" to make "sales." That took three operations, and indeed it is not possible to transform "alien" to "sales" with fewer than three operations.

Too bad! I love sales.

Our objective is to write a function called `distance` that will take two strings as arguments; we'll call them `first` and `second`. Our solution will use recursion, so we'll start with the base case. Since there are two arguments, we should expect that there will be two base cases and that they will be the "easy" or "extreme" cases.

One extreme is that one (or both) of the strings are empty. For example, imagine that `first` is the empty string. For the moment, let's assume that `second` is not empty—for example it might be "spam". Then the distance between the two strings must be the length of `second` since we must insert that many letters into the empty string to get to `second`. Similarly, if `second` is empty then the distance must be the length of `first`, since we must delete that many symbols from `first`. So, let's start our function accordingly:

```
def distance(first, second):
    '''Returns the edit distance between first and second.'''

    if first == '':
        return len(second)
    elif second == '':
        return len(first)
```

We're not done yet, but let's pause here and ask ourselves what would happen in the case that both strings were empty. In this case, the distance should be 0; notice that this is what we'll get since `first` is empty and we'll get the length of `second` —which is 0. It's always good to check these kinds of special cases to make sure we haven't missed anything.

Now for the recursion. If *first* and *second* begin with the same symbol it's pretty clear that we should consider ourselves fortunate and not mess with those matching characters. In

this case, the distance between the two strings is just the distance between the two strings with the first symbol of each sliced off. That is, we want the distance between `first[1:]` and `second[1:]`. For example, when computing the distance between *spam* and *spim* (notice that the distance is 1), the fact that they begin with the same letter lets us conclude that the distance is going to be the same as the distance between *pam* and *pim*. So, we can make a recursive call `distance(first[1:], second[1:])` .

On the other hand, if the two strings begin with different letters, the problem is more interesting. Since the first letters don't match, some sort of change will be required. We can either change the first symbol of the first string to match the first symbol of the second (a substitution), remove the first symbol of the first string (a deletion), or add a new symbol at the very front of the first string (an insertion). We don't know which of these is best, so we'll let recursion explore all three options for us. This is like the "use it or lose it" strategy but now we have *three* choices to consider rather than two.

Let's consider each of the three options. If we are going to perform a substitution, it's not hard to see that we should change the first letter in `first` to be the same as the first letter in *second*. Now these letters will match and we can remove those two letters from further consideration. Therefore, the best solution that begins with a substitution will have cost `1 + distance(first[1:], second[1:])` because the number of operations will be 1 (the substitution) plus however many operations are required to get from the remaining string `first[1:]` to the remaining string `second[1:]`.

The second option is deleting the first symbol in `first`. That's one operation, and now we would need to find the distance from `first[1:]` to `second`.

Lastly, we need to consider inserting a new symbol to the front of `first`. It's not hard to see that the new symbol should match the first symbol in `second`. That will require one operation, and then the remaining problem is to find the distance from `first` to `second[1:]`. That's because we haven't yet used the first symbol in `first`, but we've just matched the first symbol in `second`.

The best of these three options will be our best solution! Putting it all together, here's our function:

```
def distance(first, second):
    '''Returns the edit distance between first and second.'''

    if first == '':
        return len(second)
```

```
    elif second == '':
        return len(first)
    elif first[0] == second[0]:
        return distance(first[1:], second[1:])
    else:
        substitution = 1 + distance(first[1:], second[1:])
        deletion = 1 + distance(first[1:], second)
        insertion = 1 + distance(first, second[1:])
        return min(substitution, deletion, insertion)
```

Like `max`, `min` is built in to Python and returns the minimum of all of its arguments.

Wow! That's a remarkably short program that solves a challenging and important computational problem. Here are a few examples of us using this function.

```
>>> distance('spam', 'poems')
4
>>> distance('alien', 'sales')
3
```

We encourage you to try it for yourself!

```
def distance(first, second):
    '''Returns the edit distance between first and second.'''

    if first == '':
        return len(second)
    elif second == '':
        return len(first)
    elif first[0] == second[0]:
        return distance(first[1:], second[1:])
    else:
        substitution = 1 + distance(first[1:], second[1:])
        deletion = 1 + distance(first[1:], second)
        insertion = 1 + distance(first, second[1:])
        return min(substitution, deletion, insertion)

```

Run

2.12 Conclusion

This chapter has led us from the basics of Python to writing powerful recursive functions. This style of programming—using functions and recursion—is called *functional programming*.

Amazingly, what we've seen so far is enough to write *any* possible program. To be a bit more precise, the Python features that we've seen here are sufficient to write *any* program that can be written in *any other language*. You may wonder how we dare make such a bold assertion. The answer is that there is a lovely part of computer science called *computability theory*, which allows us to actually prove such statements. We'll see some aspects of computability theory in Chapter 7.

I tried Googling "recursion". It came back and said "Did you mean: recursion"

In spite of the fact that we are now functional (in both senses of

the word) programmers, there are some beautiful ideas that will allow us to write more efficient, succinct, and elegant programs, and we'll see some of these ideas in the next chapter. But before going there, you'll probably want to write some programs of your own to become comfortable with recursion.

In the meantime, we leave you with this quip from a former student of ours: "To understand recursion, you must first understand recursion."

Chapter 3: Functional Programming, Part Deux

Whoever said that pleasure wasn't functional?

—Charles Eames

3.1 Cryptography and Prime Numbers

I'm headed home soon, but there's just a bit more shopping to be done first!

Imagine that our alien is sitting at a cafe, surfing the Internet, sipping a triple mochaccino, and is now about to purchase the Harry Potter Complete DVD Collection (movies 1 through 17) from the massive online store, Nile.com. As it types in its credit card number to make its purchase, it suddenly pauses to wonder how its financial details will be kept secure as they are transmitted over the cafe's Wi-Fi and then over the vast reaches of the Internet. That's a valid concern. The good news is that many online stores use cryptography to keep such transactions secure.

Rivest, Shamir, and Adleman received the Turing Award—the computer science equivalent of the Noble Prize.

One of the most famous and widely-used cryptography schemes is called RSA, after the three computer scientists who invented it: Ron Rivest, Adi Shamir, and Leonard Adleman. Here's how it works: The online store has its own mathematical function that all customers use to encrypt their data before transmitting it over the network. This mathematical function is made publicly available. The hope is that while anyone can easily use this function to encrypt data, only the online store can "undo" (or, more technically, "invert") the function to decrypt the data and recover the original number.

For example, imagine that Nile.com tells customers to use the function $f(x) = 2x$. It's certainly easy to encrypt any number we wish to send—we simply double it. Unfortunately, any first grader with a calculator can decrypt the message by simply dividing it by 2, so that encryption function is not secure.

The RSA scheme uses a slightly more complicated function. If x is our credit-card number, we encrypt it using the function $f(x) = x^e \bmod n$, where e and n are carefully chosen numbers. (Remember from the previous chapter that $x^e \bmod n$ means

the remainder when x^e is divided by n; it can be easily computed in Python using the expression `(x**e) % n`.)

It turns out that if e and n are chosen appropriately, the online store will be able to decrypt the number to retrieve the credit card number x but it will be nearly impossible for anyone else to do so—even though everyone knows e and n.

That's pretty interesting, but how do we choose e and n and how will the store later decrypt the number that it receives? Well, we first choose two different large prime numbers p and q at random. Next, n is just pq. Now, to get the number e, we have to perform two steps: first we let $m = (p-1)(q-1)$, and then we choose our exponent e to be a random prime number less than m that is also not a divisor of m. That's it!

Now, any number x less than n can be encrypted by computing $x^e \bmod n$. Once we have selected e and n, we can share those values with anyone wishing to send us encrypted information. In a typical Internet shopping transaction, your web browser would get the publicly available values of e and n from the online store and use them to encrypt your credit card number, x. Together, the values e and n are called the *public key* for this store. (In cryptology, a public key is simply a key that can be safely published without giving away the corresponding *secret key*.)

For example, let $p = 3$ and $q = 5$. They're certainly prime (although they are way too small to be secure in practice). Now, $n = 3 \times 5 = 15$ and $m = (3-1) \times (5-1) = 8$. For our encryption exponent e, we could choose the prime number 3 because it's less than 8 and also doesn't divide 8. Now, we can encrypt any number less than n. Let's encrypt the number 13, for example. We can compute $13^3 \bmod 15$ in Python as `(13**3) % 15`; the result is 7. So 7 is our encrypted number, which we send over the Internet to the online store.

How does the store decrypt that 7 and discover that the original number was actually 13? At the same time that the encryption exponent e was computed, we should have also computed a decryption exponent d, which has two properties: it is between 1 and $m-1$, and $ed \bmod m = 1$. It's not hard to show that, because of the way e and m were chosen, there is exactly one value that has these properties; we call d the *multiplicative inverse of e modulo m*. In our example, $e = 3$ and $m = 8$, and d is also 3 (it's a coincidence that e and d are equal; that's not normally the case—if it were, the decryption key wouldn't exactly

I think the mathematics of cryptography should be called "discreet" math.

be a secret!). Notice that $ed \bmod 8 = 9 \bmod 8 = 1$. Now, the online store can decrypt any number y that it receives by simply computing $y^d \bmod n$. In our case, we received the encrypted number $y = 7$. We compute $7^3 \bmod 15$ using Python `((7**3) % 15)` and get the answer 13. Indeed, that's the value that we encrypted a moment ago! Keep in mind that while the encryption key e and n are public, the online store must keep the *decryption key* d private. Anyone with access to d can decrypt any message sent with the encryption key.

Exactly *why* this works is not too hard to show and is often taught in an introductory discrete math or algorithms course. But you may be wondering why we are so confident that the scheme is secure. This, too, requires a bit more time to explain than we have here. We will point out, however, that since the the values e and n are public, if a malicious person could find the two primes p and q that we originally selected, then they could figure out m and then d and they could crack the code. The good news is that "factoring" a number n into its prime divisors is known to be a "computationally hard" problem—*very* hard. (*Computationally hard* means a problem that takes a long time to compute the answer for.) For example, the U.S. National Institutes of Standards and Technology estimates that if we encrypt a message today using public keys that are about 600 digits long, it would take until about the year 2030 to crack the code—even if a very large number of very fast computers were used for the attack.

3.2 First-Class Functions

I prefer to do everything first-class.

Recall that in the previous chapter we learned about Python functions and explored the power of recursion. The style of programming that we examined—programs constructed from functions that call one another (and possibly themselves)—is called *functional programming*. Interestingly, in functional programming languages like Python, functions are actually data just like numbers, lists, and strings. We say that functions are "first-class citizens" of the language. In particular, we can write functions that take *other functions* as arguments and return *other functions* as results! In this chapter we'll explore these ideas, first using them to write a short program that efficiently generates long lists of primes, and ultimately writing a function that generates both the encryption and decryption functions for RSA cryptography. By the end of this chapter, we'll have written Python programs that will allow you to securely send data to your friends.

3.3 Generating Primes

Motivated by RSA cryptography, our first mission is to find a way to generate a list of primes. One reasonable way to do this is to first write a function that determines whether or not its argument is prime. Once we have such a function, we could use it to test a sequence of consecutive numbers for primality, keeping those that are prime in a list.

But how do we test if a single positive integer n is prime? One idea is to simply test whether any number between 2 and $n-1$ divides it. If so, the number is not prime. Otherwise the number is prime. (In fact, it suffices to test just the numbers between 2 and $\sqrt{n}$, since if n is not prime, at least one of its divisors must be less than or equal to $\sqrt{n}$. But for now, let's simply test all the possible divisors between 2 and $n-1$.)

To that end, it would be useful to have a function `divisors(n)` that accepts our number `n` (which we wish to test for primality) and returns `True` if `n` has any divisors (other than 1 and itself), and `False` otherwise. Actually, it will turn out to be handy if `divisors` accepts two additional numbers, `low` and `high`, that give a range of divisors to test. That will let us reduce the amount of work that has to be done.

For example, `divisors(15, 2, 14)` should return `True` because 15 has a divisor between 2 and 14 (and therefore is not prime), but `divisors(11, 2, 10)` should return `False` because 11 has no divisors between 2 and 10 (and therefore is prime). Also, note that `divisors(35, 2, 4)` should return `False` even though 35 is *not* prime.

We can write the `divisors(n, low, high)` function using recursion! To simplify matters, we'll assume that the arguments are all positive integers. If `low` is higher than `high`, then the answer is `False` because `n` cannot have any divisors in the specified range (since there are no numbers in increasing order between `low` and `high`). So, if `low > high` we must return `False`—which is a base case.

But what if `low` is less than or equal to `high`? In this case, we can test whether `n` has a divisor between `low` and `high` like this: If `n` is divisible by `low`, then we've found a divisor in that range and we must return `True`. Otherwise, the answer to the question: "Does `n` have a divisor between `low` and `high`?" now becomes the same as the answer to the question "Does `n` have a divisor between `low+1` and `high`?" But that's a version of the original question, and thus one that we can solve recursively! Here's our solution:

```
def divisors (n, low, high):
    '''Returns True if n has a divisor in the range from low to high.
```

```
    Otherwise returns False.'''
    if low > high:
        return False
    elif n % low == 0: # Is n divisible by low?
        return True
    else:
        return divisors (n , low + 1, high)
```

Now we can test if n is prime by checking whether it has any divisors between `2` and `n-1`:

```
def isPrime (n):
    '''For any n greater than or equal to 2,
    Returns True if n is prime. False if not.'''
    if divisors (n, 2, n-1):
        return False
    else :
        return True
```

We can do this even more elegantly this way:

```
def isPrime (n):
    '''For any n greater than or equal to 2,
    Returns True if n is prime. False if not.'''
    return not divisors (n, 2, n-1)
```

Recall from Chapter 2, that `not` "negates" a Boolean, so if `divisors(n, 2, n-1)` is `True` then `not divisors(n, 2, n-1)` is `False`, and if `divisors(n, 2, n-1)` is `False` then `not divisors(n, 2, n-1)` is `True`.

Now we can use `isPrime` to generate lists of primes, again using recursion. Imagine that we want to know all of the primes from 2 to some limit, for example 100. For each number in that range, we could test if it's prime and, if so, add it to our growing list of primes. Before you look at the code below, see if you can determine the base case and the recursive step for such a function.

Now, here is the Python implementation:

```
def listPrimes (n, limit):
    '''Returns a list of prime numbers between n and limit.'''
    if n == limit:
        return []
    elif isPrime (n):
        return [n] + listPrimes (n+1, limit)
```

```
    else:
        return listPrimes (n+1, limit)
```

Notice that in the second return statement, we returned `[n] + listPrimes(n+1, limit)` rather than `n + listPrimes(n+1, limit)`. Why? Well, in this case the plus sign means that two lists should be concatenated, so the expressions on its left and right have to be lists. Indeed, the result of calling `listPrimes` will be a list, since by definition a list is what this function returns (and notice that in the base case it returns the empty list). However, `n` is a number, not a list! To make it a list, we place it inside square brackets. That way, we are concatenating two lists and life is good.

The above strategy for generating primes works, but it's quite slow—particularly when attempting to generate large primes. The problem is that it repeats a lot of work. For example, if you call `listPrimes(51, 2, 50)` it will test 2 as a divisor and fail—but it will still insist on testing 4,6,8,...,50 even though we've already proven that 51 isn't even!

It's not entirely clear that Eratosthenes actually discovered this idea.

A much faster algorithm for generating primes is the so-called *sieve of Eratosthenes*. This method is named after Eratosthenes, an ancient Greek mathematician who lived around 2200 years ago. Here's the idea: To find all of the primes from 2 to 1000, for example, we first write down all of the integers in that range. Then we start with 2; it's the first prime. Now, we remove (or "sift") all multiples of 2 from this list since they are definitely not prime. When we're done, we come *back* to the beginning of our remaining list. The number 3 survived the sifting of numbers that was performed by 2, so 3 is prime. We now cancel out all remaining numbers that are multiples of 3, since they too cannot be prime. When we're done, we look at the first number in the remaining list. It's not 4 (it got sifted out earlier when we looked at 2), but 5 is there. So 5 is prime and we let it sift all of its multiples that remain in the list. We continue this process until every remaining number has had a chance to sift the numbers above it.

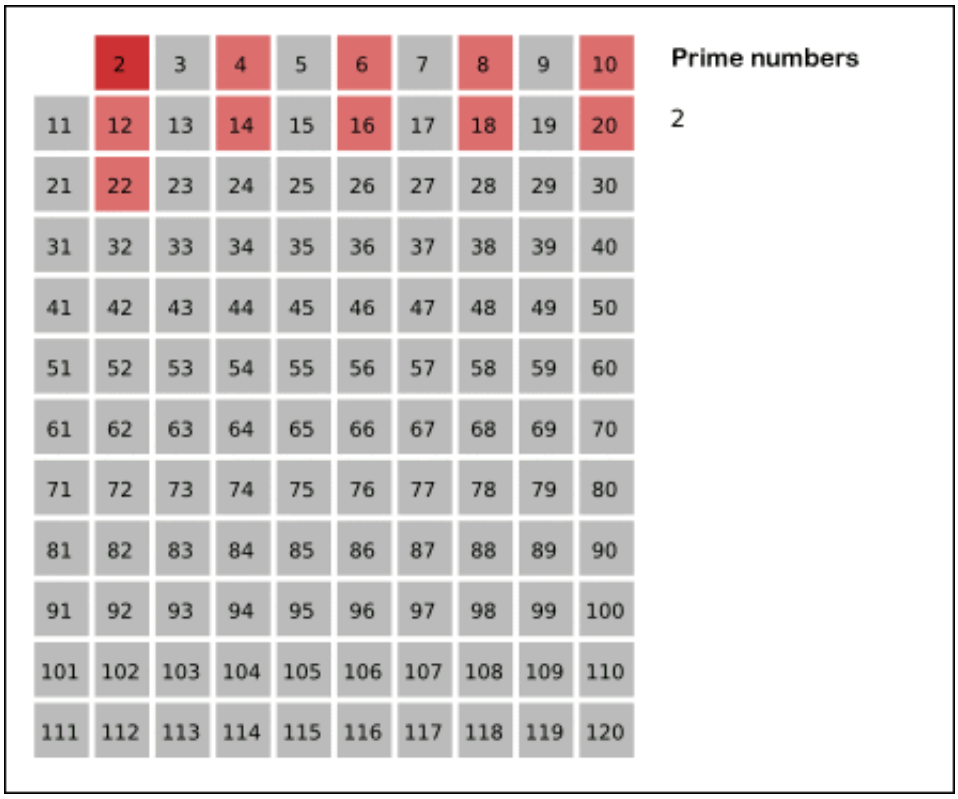


Figure 3.1: Sieve of Eratosthenes

(http://commons.wikimedia.org/wiki/File:Sieve_of_Eratosthenes_animation.gif)

Let's implement Eratosthenes' algorithm as a Python function called... `primeSieve`. It will take a list of numbers 2,3,... up to the largest number that we're interested in, and will return a list of all the primes in the original list. Fortunately, Python has a built-in function that allows us to obtain the list of all integers from a starting point to an ending point. It's called `range`, and in Python 2 it works like this:

```
>>> range(0,5)
[0, 1, 2, 3, 4]
>>> range(3,7)
[3, 4, 5, 6]
```

In Python 3 `range` works almost the same, but it needs a little nudge to turn the result into a list:

```
>>> list(range(0,5))
[0, 1, 2, 3, 4]
>>> list(range(3,7))
[3, 4, 5, 6]
```

Notice that the list that we get back from `range` seems to stop one number too soon. That may seem weird, but as we'll see later, it turns out to be useful. So getting *all* the integers from 2 to 1000, for example, is easy: `range(2, 1001)`. We can then pass that list into the `primeSieve` function.

Before writing primeSieve let's just do a small thought experiment to better understand how it will work. Imagine that we start it with the list `range(2, 11)`, which is:

```
[2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Passing this list to `primeSieve` is basically saying "Could you please find me all of the primes in this list?" To accommodate your polite request, the `primeSieve` function should grab the 2 and hold on to it because it's a prime. It should then sift all of the multiples of 2 from this list, resulting in a new list:

```
[3, 5, 7, 9]
```

Now what? Well, `primeSieve` would like to do as little work as possible and instead send that list to some function and ask it, "Could you please find me all of the primes in *this* list?" Aha! We can send that list *back* to `primeSieve` because its job is to find the primes in a given list. That's just recursion!

So, continuing with our example, the first time we called `primeSieve` it found the 2, sifted out the multiples of 2 to get the list `[3, 5, 7, 9]`, and called `primeSieve` on that list. Whatever comes back from the recursive call will be tacked on to the 2 that we're currently holding—and then we'll have the whole list of primes from 2 to 10.

The recursive call to primeSieve with the argument list `[3, 5, 7, 9]` will similarly grab the 3 from the front of that list, sift out all of the multiples of 3 to get list `[5, 7]`, and will ask for help finding the primes in that list. Whatever primes are returned will be tacked on to the 3 that we're holding, and that will be all of the primes in the list `[3, 5, 7, 9]`.

In the next recursive call, we'll grab 5 and recur on the list `[7]`. *That* recursive call will grab the 7 and recur on the empty list. We see here that since the list is getting shorter each time, the base case arises when we have an empty list. In that case, `primeSieve` must report that "the list of all the primes in my argument is empty"—that is, it should return the empty list.

I think that this approach should be called the "wishful thinking" method

We still need a function that will do the actual sifting. We can

imagine a function called `sift` that takes two arguments—a number `toRemove` and a list of numbers `numList`—and returns all of the numbers in `numList` that are *not* multiples of `toRemove`. Let's come back to that in a moment, and write our `primeSieve` under the assumption that we have `sift`. This approach for writing programs is called *top-down design* because we start at the "top" (what we want) and then work our way down to the details that we need.

because we just wish for a helper function whenever we need one.

```
def primeSieve(numberList):
    '''Returns the list of all primes in numberList, using a prime sieve
    if numberList == []:      # if the list is empty,
        return []             # ...we're done
    else:
        prime = numberList[0]  # The first element is prime!
        return [prime] + primeSieve(sift(prime, numberList[1:]))
```

Now, we need to sift. The next section will introduce a tool that will help us do exactly that, so read on...

3.4 Filtering

Fortunately for us, Python (like most functional programming languages) has a built-in function named `filter` that does (almost) exactly the sifting that we would like to do.

We'll eventually get back to the problem of general list sifting that we started above, but for the moment let's just focus on the problem of sifting a list by removing numbers divisible by two. To demonstrate `filter` in action, let's first define a function called `isNotDivisibleBy2` that takes a number `n` as an argument and returns a Boolean value: `True` if the number is not divisible by 2 (i.e., it is odd) and `False` otherwise (i.e. it is even).

```
def isNotDivisibleBy2(n):
    '''Returns True if n is not divisible by 2,
    else returns False.'''

    return n % 2 != 0
```

Now back to filtering. Having `isNotDivisibleBy2`, here's how we can use it with Python's `filter` function. In Python 2 it looks like this:

```
>>>filter(isNotDivisibleBy2, range(3, 10))
[3, 5, 7, 9]
```

In Python 3, `filter` doesn't eagerly produce a list, so we need to send its result to the `list` function, as follows:

```
>>> list(filter(isNotDivisibleBy2, range(3, 10)))
[3, 5, 7, 9]
```

I couldn't function without a filter to remove the noxious oxygen from Earth's air!

You may have inferred what `filter` is doing: its first argument is a *function* and its second argument is a list. The function is a special one that takes a single argument and returns a Boolean result. A function that returns a Boolean is called a *predicate*; you can think of the predicate as telling us whether or not it "likes" its argument. Then, `filter` gives us back all of the elements in the list that the predicate likes. In our example, `isNotDivisibleBy2` is a predicate that likes odd numbers. So we got back the list of all the odd numbers in the original list.

A function that takes other functions as arguments?! Strange, but definitely allowed, and even encouraged! This idea is central to functional programming: functions can be passed into and returned from other functions just like any other kind of data. In fact, Chapter 4 will explain why this idea is not so strange after all.

Four letr wrds rock!

Lists of numbers are not all we can filter. Here's another example of `filter`, this time using lists of strings. In preparation for its next visit to Earth, our alien is attempting to master English and a number of other languages. The alien has a list of words to learn, but it's particularly keen on learning the four-letter words first. For example, if it's given the list of words `['aardvark', 'darn', 'heck', 'spam', 'zyzzyva']` it would like to have a way of filtering that list to just be the "bad" words `['darn', 'heck', 'spam']`.

Again, we define a predicate function first: a function called `isBad` that takes a string named `word` as an argument and returns a Boolean value: `True` if the word is of length four and `False` otherwise.

```
def isBad(word):
```

```
    '''Returns True if the length of "word" is 4, else returns False.'''
    return len(word) == 4
```

Now that we have `isBad`, here's how we can use it with Python's `filter` function:

```
>>> filter(isBad, ['ugh', 'darn', 'heck', 'spam', 'zy
['darn', 'heck', 'spam']
```

Again, in Python 3, we need to send the result of `filter` to the `list` function to actually get the list, so the line would begin `list(filter(bad,...)`.

3.5 Lambda

`Filter` helps us sift lists of numbers, but so far all we've seen how to do is to sift them to remove even numbers. If we wanted to remove multiples of 3 we would need another helper function:

```
def isNotDivisibleBy3(n):
    '''Returns True if n is not divisible by 3, else returns False.'''
    return n % 3 != 0
```

And then we'd need another to remove multiples of 5, and one for 7, and 11, and so on. Obviously this is not going to work.

The idea behind our `primeSieve` function is that we want to change what we are filtering for "on the fly," depending on which number is currently first in the list. You might imagine that we could add a second argument to our predicate function, as follows:

```
def isNotDivisibleBy(n, d):
    '''Returns True if n is not divisible by d, else returns False.'''
    return n % d != 0
```

Generally, this would solve the problem, but in this case we have a problem: `filter` *requires* that the predicate function we pass it takes only one argument. So this new definition will not work either.

What we really need is a sort of "disposable" function that we can define just when we need it—using the number we currently care about sifting—and then immediately throw it away after we are

done using it.

Is a disposable function environmentally friendly?!

In fact, this function will be so temporary that we won't even bother to give it a name! Such a function is called an anonymous function.

Here's what an anonymous function definition looks like for our `isNotDivisibleBy2` example:

```
>>> filter(lambda n: n % 2 != 0, range(0, 1001))
```

The word "lambda" indicates that we are defining an anonymous function—a short-lived function that we need right here and nowhere else. Then comes the names of the arguments to the function (in this case, one argument named `n`), then a colon, and then the value that this function should return. So, this anonymous function `lambda n: n % 2 != 0` is exactly equivalent to our `isNotDivisibleBy2` function above.

"Lambda" derives from a branch of mathematics called the lambda calculus (see section 3.7), *which influenced the first functional language, LISP.*

The anonymous function syntax in Python is odd; in particular, we don't put parentheses around the function's arguments, and there is no `return` statement. Instead, anonymous functions in Python implicitly return whatever value comes after the colon.

Just to make the point that anonymous functions really are full-fledged functions, take a look at this:

```
1  double = lambda x: 2 * x
2  print (double(21))
```

<< First < Back Step 1 of 4 Forward > Last >>

line that has just executed
next line to execute

Frames Objects

(ch03_lambda)

Whoa—that's really weird! In the first line, we're defining a variable named `double` and giving it the value `lambda x: 2 * x`. But in a functional programming language functions are truly *first-class citizens*; they are data just like numbers, strings, and lists. So `double` is a function with one argument, and if we want to use it we need to pass it a value. In the second line, that's exactly what we're doing. In fact, when we define a function in the "normal" way, that is by starting with the line `def double(n)`, we are *really* just saying `double` is a variable whose value is the function that I'm going to define in the lines following the `def` statement."

Finally, we can use anonymous functions to finish writing `sift`, as follows:

```
def sift(toRemove, numList):
    '''Takes a number, toRemove, and a list of numbers
    Returns the list of those numbers in numList that

    return filter(lambda x: x % toRemove != 0, numList
```

If using Python 3, remember that we need to place the `filter` *call inside* `list(...)`.

The anonymous function we pass into `filter` uses `toRemove` in its body without having to pass it in as an argument. It can do this because this function is defined in an environment where `toRemove` *already exists* and has a value.

Of course we could also write `sift` using the list-comprehension syntax (see the sidebar in section 3.6.2); then it would look like this:

```
def sift(toRemove, numList):
    return [x for x in numList if x % toRemove != 0]
```

Finally, as a reminder, here is our `primeSieve` function again, using `sift`:

```
def primeSieve(numberList):
    '''Returns the list of all primes in numberList using a prime sieve a
    if numberList == []:      # if the list is empty,
        return []              # ...we're done
    else:
        prime = numberList[0]  # The first element is prime!
        return [prime] + primeSieve(sift(prime, numberList[1:]))
```

It's surprising how much entertainment value there is in running `primeSieve` to generate long lists of primes. However, Python got angry with us when we tried something like this:

If using Python 3, this will be `primeSieve(list(range(2, 10000)))`.

```
>>> primeSieve(range(2, 10000))
```

We got a long and unfriendly error message. The reason is that this created a lot of recursion, and Python is trained to believe that if there is a lot of recursion going on, there must be an error in your program (usually because you forgot to put in a base case). Exactly how much recursion Python thinks is too much depends on the Python version and your computer's operating system. However, you can ask Python to allow you more recursion by including the following two lines at the top of your file:

```
import sys
sys.setrecursionlimit(20000) # Allow 20000 levels of recursion
```

We asked for $20,000$ levels of recursion here. Some operating systems may allow you more or less than this. (Most modern machines will allow much, much more.)

This brings us to one last point: while the prime sieve is quite efficient, most good implementations of the RSA scheme use even more efficient methods to generate large prime numbers. In practice, the primes used by RSA when encoding Internet transactions are generally a few hundred digits long! A CS course on algorithms or cryptography may well show you some of the more sophisticated and efficient algorithms for generating primes.

Shameless sales pitch!

3.6 Putting Google on the Map!

In the rest of this chapter we will continue to build on the idea of functions as first-class citizens that can be passed into or returned from other functions. We'll look at two more examples: Google's MapReduce approach to processing data, and taking derivatives of functions.

Imagine that you work for Nile.com. Your company maintains a list of product prices, and

periodically the prices are systematically increased. Your boss comes into your office one morning and asks you to write a function called `increment` that takes a list of numbers as an argument and returns a new list in which each number is replaced by a value one greater. So, with the argument `[10, 20, 30]`, the result should be `[11, 21, 31]`.

No problem! We can write `increment` recursively. For the base case, if the argument is an empty list, we'll also return an empty list. (Remember, the result is always a list of the same length as the argument, so returning anything other than the empty list in this case would be violating our contract!). For the recursive case, we observe that we can easily increment the first number in the list: We simply find that element and add one to it. We can then slice that first element off the list and increment the remainder. The phrase "increment the remainder" is essentially saying "use recursion on the remaining list." In other words, `increment([10, 20, 30])` will first find the 10, add one to make it 11, and then recursively call `increment([20, 30])` to get the result `[21, 31]`. Once we have that, we just concatenate the 11 to the front of that list, resulting in `[11, 21, 31]`. Here's the Python program:

```
def incrementList(numberList):
    '''Takes a list of numbers as an argument and returns
       a new list with each number incremented by one.'''

    if numberList == []:
        return []
    else:
        newFirst = numberList[0] + 1  # increment 1st element
        # Next, increment the remaining list
        incrementedList = incrementList(numberList[1:])
        # Now return the new first element and the
        # incremented remaining list
        return [newFirst] + incrementedList
```

3.6.1 Map

Your boss is pleased, but now tells you "I need a very similar function that takes a list as an argument and adds 2 to each element in that list." Obviously, we could modify `increment` very slightly to do this. Then your boss tells you, "I need yet another function that takes a list as an argument and triples each element." We can do that too, but this is getting old.

This boss is straight out of Dilbert!

We see that your boss frequently needs us to write functions that take a list of numbers

as an argument and return a new list in which *some function* is applied to every element in that list.

An important principle in computer science is "if you are building lots of very similar things, try instead to build one thing that can do it all."

This is known more succinctly as the principle of generalization. *We're trying to make our function more general and less specific.*

We've seen this principle before when we started writing functions, but here we'll take it one step further. The "do it all" thing in this case is a function called `map`, which is built in to Python and many other functional programming languages. We'll show it to you and then explain. But first, it will be handy to have two simple functions to help with our example. One function, called `increment`, takes a number as an argument and returns that number plus 1. The second, called `triple`, takes a number as an argument and returns three times that number:

```
def increment(x):
    '''Takes a number x as an argument and returns x + 1.'''
    return x+1
```

```
def triple(x):
    '''Takes a number x as an argument and returns 3 * x.'''
    return 3 * x
```

If you're using Python 2, you can now do the following:

```
>>> map(increment, [10, 20, 30])
[11, 21, 31]
>>> map(triple, [1, 2, 3, 4])
[3, 6, 9, 12]
```

If you're using Python 3, `map` is a bit lazy and requires some cajoling to produce the list, just like `filter` did. The magic incantation required in Python 3 looks like this:

```
>>> list(map(increment, [10, 20, 30]))
[11, 21, 31]
>>> list(map(triple, [1, 2, 3, 4]))
[3, 6, 9, 12]
```

This is passing the result of `map` to a built-in function called `list`, which forces Python 3 to actually produce the list.

Notice that `map` takes two arguments. The first is a function; the second is a list. The function that is given to `map` (e.g., `increment` and `triple` in our examples) **must** be a function that takes one argument and produces a single result. Then, `map` applies that function one-by-one to each element in the list to build a new list of elements.

Sometimes, we don't need to create our own functions to give to `map`. For example, take a look at this example:

```
>>> map(len, ['I', 'like', 'spam'])
[1, 4, 4]
```

In Python 3, this would be

```
>>> list(map(len, ['I', 'like', 'spam']))
[1, 4, 4]
```

In this case, the built-in function `len` is applied to each of the strings in the given list. First, `len` is applied to the string `'I'`. The length of the string is 1 and that's the first thing to go in the result list. Then the `len` function is applied to the string `'like'`; the result is 4, which is the next thing to go in the result. Finally, the length of `'spam'` is 4, which is the last value placed in the result list.

Notice how our abstraction theme comes into play with `map`. The details of how the function we pass to `map` is applied to the list are hidden away. We no longer have to worry specifically about how to step through the list to modify each element: `map` does this for us.

List Comprehensions

The `map`, `reduce`, and `filter` functions, and the `lambda` notation for anonymous functions, are not unique to Python; they are found in most functional programming languages. However, Python offers an alternative syntax to `map` and `filter` called *list comprehensions*, which allow us to avoid using `lambda`. Sometimes that's both conceptually easier and faster to type!

Let's start with the example from Section 3.6.1, where we wanted to increment every element in the list `[10, 20, 30]`. Whereas earlier we used `map` and an `increment` function to do this, the list comprehension syntax looks like this:

```
>>> [x + 1 for x in [10, 20, 30]]
[11, 21, 31]
```

Similarly, we can triple every element in the list `[1, 2, 3, 4]` using the syntax:

```
>>> [3 * x for x in [1, 2, 3, 4]]
[3, 6, 9, 12]
```

In general, the syntax is:

```
[f(x) for x in L]
```

...where `f` is some function and `L` is some list. Notice that this is really just like `map` in that we are mapping the function `f` to every element `x` in the list `L`. By the way, there's nothing special about the name `x` for the variable; we could use a different variable name as in:

```
>>> [len(myString) for myString in ['I', 'like', 'spam']]
[1, 4, 4]
```

A very similar syntax can be used to do the job of `filter`. Rather than using `filter` to obtain the four-letter words in a list as we saw earlier, we can use list comprehensions this way:

```
>>> words = ['aardvark', 'darn', 'heck', 'spam', 'zyzzyva']
>>> [x for x in words if len(x) == 4]
['darn', 'heck', 'spam']
```

In this case, we defined the list `words` before using the list comprehension, just to make the point that we can do that. Of course, we could have also done this as:

```
>>> [x for x in ['aardvark', 'darn', 'heck', 'spam', 'zyzzyv
... if len(x) == 4]
```

Similarly, we could get the multiples of 42 between 0 and 1 million (another example where we used `filter`) this way:

```
>>> [x for x in range(0, 1000000) if x % 42 == 0]
```

In general, the format for using list comprehensions to filter looks like this:

```
[x for x in L if ...]
```

...where `L` is a list and the ... represents some Boolean expression; that is, an expression that evaluates to either `True` or `False`. We get back the list of all values of `x` in the list `L` for which that Boolean expression (recall that it's called a predicate) is `True`. That is, we get all of the values of `x` that the predicate "likes." Finally, we can write list comprehensions that combine both `map` and `filter` into one. Here's an example where we produce the square of every even number between 0 and 10:

```
>>> [x**2 for x in range(0, 11) if x % 2 == 0]
[0, 4, 16, 36, 64, 100]
```

This list comprehension is mapping the function $f(x) = x^2$ to every value of `x` in the list of integers from 0 to 10 subject to filtering that list through a predicate that only likes even numbers. Thus, we get the squares of even numbers. In general, the syntax is:

```
[f(x) for x in L if ...]
```

where `f` is a function, `L` is a list, and what comes after the `...` is a predicate. We get back the list of $f(x)$ values for those values of `x` for which the predicate is `True`.

By the way, this syntax works just the same in Python 2 and 3. Python 3 doesn't require any special prodding to produce a list when list comprehensions are used.

3.6.2 Reduce

Ten mochaccinos, a laptop, the Harry Potter DVD collection,...

Our alien has just learned about `map` and is quite excited. During its visit to Earth, the alien purchased a number of items and it plans to request reimbursement from its employer.

For example, here is a list of the costs, in U.S. dollars, of several items purchased by the alien: `[14, 10, 12, 5]`. The alien would like to convert these costs to its own currency and then add up the total. The exchange rate is 1 U.S. dollar equals 3 alien dollars. So, the values of the four items in alien dollars are `[42, 30, 36, 15]` and the total request for reimbursement in alien dollars will be $42 + 30 + 36 + 15 = 123$.

Converting a list from U.S. dollars to alien dollars is no problem—we did that above using `map` and our `triple` function. Now, though, we want to add up the elements in the resulting list. We could write a recursive function for that task. However, it turns out that there are many very similar tasks that the alien needs to perform as well.

Here's one: Imagine that the alien converted between a variety of currencies while shopping on Earth. For example, the alien first got some dollars, then changed dollars into Euros, then changed Euros into rubles, and then rubles into yen. If the alien started with 1 U.S. dollar, how many yen is that? This is a matter of computing the products of exchange rates. If one dollar is 0.7 Euros, one Euro is 42 rubles, and one ruble is 3 yen, then one dollar is $0.7 \times 42 \times 3 = 88.2$ yen. So now the alien needs to compute the *product* of a list of numbers.

Again, rather than writing a separate function to add up the numbers in a list and another to multiply the numbers in a list (and possibly others to do other things to elements in a list), Python provides a general-purpose function called `reduce` that reduces all of the elements in a list to a single element (e.g., their sum or product or something else).

Let's first define a function called `add` that takes *two* arguments and returns their sum, and another called `multiply` that takes *two* arguments and returns their product.

```
def add(x, y):
    '''Returns the sum of the two arguments.'''
    return x + y

def multiply(x, y):
    '''Takes two numbers and returns their product.'''
```

```
    return x * y
```

Now, here's `reduce` in action:

```
>>> reduce(add, [1, 2, 3, 4])
10
>>> reduce(multiply, [1, 2, 3, 4])
24
```

In the first case `reduce` reduced our list to the sum of its elements, and in the second case to the product of its elements. Notice that the first argument to `reduce` is a function and the second is a list. A subtle point here is that the function that we give as an argument to `reduce` must take *two* arguments and return a single result. Then, `reduce` takes the first two elements from the list and applies the given function to reduce them to one element, takes that result and the next element from the list and applies the function to them, and repeats that process until all of the elements are reduced to a single value.

3.6.3 Composition and MapReduce

Finally, imagine that we plan to frequently take a list, map some function (e.g., `triple`) to each element in that list to produce a new list (e.g., the list of 3 times each element), and then apply some other function to reduce that new list to a single value (e.g., using `add` to get the sum of these values). This is a combination of `map` and `reduce`, or more accurately the *composition* of `map` and `reduce`. Let's write such a function and call it `mapReduce`. It will take two functions and a list as arguments—the first function for `map` and the second for `reduce`:

While `reduce` *is built in to Python 2, in Python 3 you'll need to include the line* `from functools import *` *before using* `reduce`*. You can include that line at the Python prompt or at the top of any file that uses* `reduce`*.*

Remember that in Python 3 you'll need to include the line `from functools import *` *since we're using* `reduce` *here.*

```
1  def add(x, y):
2      '''Returns the sum of the two
3              arguments.'''
4      return x + y
5
6  def triple(x):
7      '''Takes a number x as an argument
8              and returns 3 * x.'''
9      return 3 * x
10
11 def mapReduce(mapFunction, reduceFunction, myL
12     '''Applies mapFunction to myList to
13             construct a new list and then
14             applies reduceFunction to the
15             new list and returns
16             reduceFunction's result.'''
17
```

<< First | < Back | Step 1 of 22 | Forward > | Last >>

line that has just executed

next line to execute

Frames Objects

(ch03_mapreduce)

Alternatively, this can be done more succinctly in one line as follows:

```
def mapReduce(mapFunction, reduceFunction, myList):
    '''Applies mapFunction to myList to construct a new list
    and then applies reduceFunction to the new list
    and returns that value.'''

    return reduce(reduceFunction, map(mapFunction, myList))
```

Now, for example, we can use `mapReduce` to convert prices from U.S. dollars to alien dollars and add up the total, like this:

```
>>> mapReduce(triple, add, [14, 10, 12, 5])
123
```

The second version of `mapReduce` above is particularly elegant in the way that `map` and `reduce` are composed or "glued" in one line. Notice that in order for Python to evaluate the expression `reduce(reduceFunction, map(mapFunction, myList))` it must first evaluate `map(mapFunction, myList)`, since that is one of the arguments to `reduce`. So, the `map` function is applied to `mapFunction` and list `myList` to get the resulting list. Now, `reduce` has both the function and the list that it needs to do its thing.

The `mapReduce` function is so broadly useful that it inspired a major part of Google's internal software design! To read more about it, you can (of course) just Google "mapReduce."

Functional Programming Meets Psychotherapy

The ideas behind functional programming actually predate the computer. A mathematical framework called the *lambda calculus* was developed by the logician Alonzo Church in the 1930's to examine the limits of what is theoretically computable. (We address some of these issues in Chapter 7.) The lambda calculus served as the basis of actual programming languages, such as the early language LISP developed at MIT in the 1950's. The "lambda" in lambda calculus is the reason for its appearance in Python's anonymous functions.

LISP and related functional programming languages have been of particular interest to researchers in Artificial Intelligence (AI) because they are well suited for analyzing text. We can take a human's typed input and write functions that process the words as strings and sentences as lists of strings. This is called *natural language processing*. (The "natural" here refers to human languages in contrast to programming languages.)

One of the early and controversial examples of natural language processing and artificial intelligence was the ELIZA program, first written by Professor Joseph Weizenbaum at MIT in the mid-1960's. Eliza impersonated a psychologist and allowed the human user to enter text that ELIZA would process and return a simple response that—while not actually aware of the user's emotions—seemed generally appropriate. Eliza was first written by

Weizenbaum in a language called SLIP, but was later reprogrammed in LISP.

Here's the beginning of an actual transcript of an interaction with ELIZA from an online version of the program (available here).

```
Eliza: Hello. I am ELIZA. How can I help you?

You:  I am learning about functional programming.

Eliza:  How long have you been learning about functional
programming?

You:  I guess it has been a few weeks now.

Eliza: Can you elaborate on that?
```

Responses such as "Can you elaborate on that?" or even "Tell me about your mother" are commonly used and quickly reveal that ELIZA is actually rather clueless. While Weizenbaum evidently only intended ELIZA to serve as a demonstration of the power of natural language processing, some people actually took an interest in using ELIZA as their therapist! A book and a documentary have been made about Weizenbaum and ELIZA, and a number of free and commercial versions of the program are available, including some that have a very bad attitude and deliberately berate their "patient."

3.7: Functions as Results

If functions can be the arguments to other functions, it must also be possible for them to be the results of functions, right? Take a look at this `scale` function and try to figure out what it does:

```
def scale(n):
    return lambda x: n * x

# Imagine that we now do this:
f = scale(42)

print (f(4))

# Run the code to see what is printed.

```

Run

The `scale` function takes the number 42 as an argument and returns a function (the `lambda` indicates that we're giving back a function) that we are then assigning to the variable `f`. So `f` is a function, but what does that function do? Well, for any argument `x` it returns `42 * x`—as we see here:

```
>>> f(2)
84
>>> f(10)
420
```

There's something odd here! In the definition of the `scale` function, the line:

```
return lambda x: n * x
```

makes it look like the function that we're returning has *two variables*, `n` and `x`, whereas the `lambda x` indicates that this a function of just *one* variable, `x`. Indeed, in the examples above, we defined `f = scale(42)` and then we gave `f` just *one* argument as in `f(2)` and

`f(10)`.

When we said `f = scale(42)`, the 42 was passed into `scale`'s argument `n`, and `n` was replaced with 42. Therefore, `scale(42)` actually returned

```
lambda x: 42 * x
```

Maybe I can write a program that will write all my future CS programming assignments for me!

That's clearly a function of just *one* variable and that is the function that we then assigned to our variable `f`.

Although the `scale` function is admittedly not super-useful, the amazing thing here is that we wrote a program that can write other programs—a powerful concept that is widely used by computer scientists. Indeed, many complex programs are, at least in part, written by other programs.

By the way, rather than using an anonymous function inside our `scale` function, we could have given the function a name and returned that function. We do this as follows:

```
def scale(n):

    def multiply(x): # Here we are defining a new function,
        return n * x # but it's INSIDE scale!

    return multiply  # Here we are returning that function
```

Notice that the indented `def multiply(x)` indicates that `multiply` is being defined inside `scale`, so this is defining `multiply` to be one of `scale`'s own variables, just like defining any variable within a function. The only difference is that `multiply` is a *function* rather than a number, list, or string. Then, once we've defined that variable, we return it. Now, calling `scale(42)`, for example, gives us back a function and we can do exactly what we did a moment ago with the first version of `scale`:

```
>>> f = scale(42)
>>> f(2)
84
>>> f(10)
420
```

Functions that take other functions as arguments or return a function as a result are called *higher-order functions*.

3.7.1: Python Does Calculus!

I'd rather impersonate them and alienate them!

In addition to learning English and other languages, our alien has been advised to study calculus before returning to Earth, so that it can better impersonate a college student.

You probably recall the idea of taking the derivative of a function. For example, the derivative of $f(x) = x^2$ is a new function $f'(x) = 2x$. For our purposes, the key observation is that the derivative of a function is another function. You may also recall that there are many rules for computing derivatives, and yet some functions are very hard to differentiate and others are downright impossible. So in some cases we may want to approximate the derivative—and we can do this computationally! In fact, let's write a Python function that differentiates *any* function (approximately).

We looked back at our own calculus notes and found that the derivative of a function $f(x)$ is a function $f'(x)$ defined as follows:

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

The derivative is defined as "the limit as h goes to zero." However, for small positive values of h, like $h = 0.0001$, we get a good approximation of the limit. We can make the approximation as good as we want by making h as small as we want. Our objective, then, is to write a function called `derivative` that takes a function and a value of h as an argument, and returns another function that is the approximation of the derivative for that value of h. Here we go:

```
def derivative(f, h):
    '''Returns a new function that is the approximation of
    the derivative of f with respect to h.'''

    return lambda x: (f(x+h) - f(x)) / h
```

Notice that we're returning a function, as we're supposed to. That function takes an argument `x` and returns the approximate value of the derivative at that value `x` for the given `h`.

Wait a second! It seems that the function that we're returning is using a function `f` and

two numeric variables, `x` and `h`—so why do we see just the variable `x` after the `lambda`? This is the same issue that we saw earlier with the `scale` function. Remember that when we call the `derivative` function, we pass it both a function `f` and a number `h`. Then, that function and that number are "plugged in" wherever we see a `f` and a `h` in the program. In the expression `lambda x: (f(x+h) - f(x)) / h`, both the `f` and the `h` are no longer variables—they are actual values (albeit the `f` is a value that happens to be a function). So, the `lambda` anonymous function truly only has one argument, and that's `x`.

Let's try it out. First, let's define a function x^2 and name it `square`.

```
def square(x):
    return x**2
```

Now, let's use our `derivative` function:

```
>>> g = derivative(square, 0.0001)
>>> g(10)
20.000099999890608
```

The actual derivative of x^2 is $2x$, so the derivative at $x = 10$ is actually 20. We're getting a pretty good approximation. That's good, but here's something better: if we now want to find the second derivative of x^2, we can differentiate our first derivative. The actual second second derivative of x^2 is simply 2 (for all values of x). Let's see what our `derivative` function says.

The second derivative is "accelerated" material!

```
>>> h = derivative(g, 0.0001)
>>> h(10)
2.0000015865662135
```

3.7.2: Higher Derivatives

Speaking of second derivatives, it sure would be handy to have a more general function that could find us the first, second, third, or generally, the k^{th} derivative of a function. Let's write a function called `kthDerivative(f, h, k)` to do this. It will take arguments that include a function `f`, a small number `h`, and a positive integer `k` and will return an approximation of the k^{th} derivative of f. We could do this "from scratch", not relying on

the existing `derivative` function that we just wrote, but since we have it already let's use `derivative`.

What's the base case? The simplest case appears to be when $k = 1$, in which case we just return the derivative of the given function. (Although it would be equally defensible to have a base case at $k = 0$, in which case we would just return f itself.) Otherwise, $k > 1$ and we need to find the recursive substructure. Well, the k^{th} derivative of f is, by definition, just the derivative of the $(k-1)^{st}$ derivative of f. So we can ask the `kthDerivative` function to find the $(k-1)^{st}$ derivative of `f` and then apply our `derivative` function to it. This should be fun and easy.

```
def kthDerivative(f, h, k):
    '''Returns a new function that is the approximati
    the kth derivative of f with respect to h.'''
    if k == 1:
        return derivative(f, h)
    else:
        return derivative(kthDerivative(f, h, k-1), h
```

A function of degree 4 is "quartic", one of degree 9 is "nonic". Believe it or not, one of degree 100 is called "hectic."

Let's take this out for a spin by defining the "quartic" function x^4 and then taking the third derivative, which we know to be $24x$.

```
def quartic(x):
    '''Returns x**4.'''

    return x**4
```

```
>>> g = kthDerivative(quartic, 0.0001, 3)
>>> g(10)
241.9255906715989
```

Please! Let's not go any further down the slippery slope *of bad math puns.*

The actual answer should have been 240; we got an approximation due to the value we used for h.

That was a high-velocity discussion of derivatives, but it demonstrated some of the integral features of functional programming.

3.8: RSA Cryptography Revisited

We began this chapter with a discussion of RSA cryptography. Recall that if Alice wants to be able to receive secure messages, then she can use the RSA algorithm to generate a public and a private key. The public key is entirely public—anyone can use it to encrypt a message to Alice. The private key, however, belongs exclusively to her; she uses it to decrypt messages encrypted using her public key. Of course, if Alice's friend, Bob, wants to receive encrypted messages too, he can use RSA to generate his own public and private keys. He then shares his public key and keeps his private key secure. When you want to send an encrypted message to Alice, you use her public key to encrypt the message. When you want to send a message to Bob, you use his public key.

Our objective is to write a function called `makeEncoderDecoder()`, which takes no arguments, constructs the RSA encryption and decryption keys, and returns *two* functions: The first encrypts data using the encryption key (which is built into the function, so we don't need to keep track of it—convenient when a key is hundreds of digits long!) The second function decrypts encrypted data using the decryption key (again built-in). Alice, Bob, you, and your friends will all be able to use the `makeEncoderDecoder()` function to construct encryption and decryption functions; each encryption function will be made public and each decryption function will be kept by its owner. Here's an example of Alice using `makeEncoderDecoder`.

```
>>> AliceEncrypt, AliceDecrypt = makeEncoderDecoder()
Maximum number that can be encrypted is 34
>>> AliceEncrypt(5)
10
>>> AliceDecrypt(10)
5
>>> AliceEncrypt(31)
26
>>> AliceDecrypt(26)
31
```

Notice here that `makeEncoderDecoder` returned two functions (we'll see in a moment how we can return two things), which we've named `AliceEncrypt` and `AliceDecrypt`. Then, we tested those two functions by encrypting them using Alice's encryption function (which contains the encryption key inside of it) and then decrypting them with Alice's decryption function (which incorporates the decryption key).

Let's first remind ourselves how the RSA scheme works. We begin by choosing two different random prime numbers p and q (preferably large, but we'll be able to decide how large later). We compute $n = pq$ and $m = (p-1)(q-1)$. We then choose the

public encryption key e to be a random prime number between 2 and $m - 1$ such that e does not divide m. Next, we construct the decryption key d to be the multiplicative inverse of e modulo m—that is, the unique number d such that $d \leq m - 1$ and $ed \mod m == 1$. Now, we can encrypt a number x between 1 and $n - 1$ by computing $y = x^e \mod n$. The value y is the encrypted message. We can decrypt y by computing $y^d \mod n$.

We've already written functions that produce lists of prime numbers. We'll use our efficient `primeSieve` function for that purpose. Recall that this function takes a list of consecutive integers from 2 to some largest value and returns the list of all primes in that range. Our first task will be to call the `primeSieve` function and choose two different prime numbers from that list. For now, let's restrict the range of the prime numbers to be between 2 and 10. This will be useful for testing purposes, and later you can change the maximum value from 10 to something much larger.

To choose items at random, we can use Python's `random` package. We tell Python that we want to use that package by including the line `import random` at the top of the file. Now, we have access to a variety of functions in that package. You can find out what's in the package by looking at the Python documentation Web site or by typing `import random` at the Python prompt and then typing `help(random)`, which will describe all of the functions in that package.

One of the most useful functions in the `random` package is `random.choice`. It takes a list as an argument and returns a randomly selected element of that list. Here's an example of that function in action:

```
>>> import random
>>> random.choice([1, 2, 3, 4])
3
>>> random.choice([1, 2, 3, 4])
2
```

We need to choose two different prime numbers in the range from 2 to 10. We could use `random.choice` twice, one to choose each of the two prime numbers, but there is a chance that we'll get the same prime number twice, which doesn't work for RSA. So, we can use another function in the `random` package, `random.sample`. It takes two arguments: a list and the number of items that we want to choose randomly—but uniquely—from that list. The function returns a list of the randomly selected items. Here's an example of `random.sample` in action:

```
>>> import random
>>> random.sample([1, 2, 3, 4], 2)   # Pick 2 items at random
[4, 3]
>>> random.sample(range(2, 100), 3)  # Pick 3 different items
[17, 42, 23]
```

When a function returns a list of several items and we know how many items will be in that list, we can give names to those items like this:

```
>>> import random
>>> a, b = random.sample([1, 2, 3, 4], 2)
>>> a
4
>>> b
3
```

In this case, we knew that the `random.sample` function was going to return a list of two items (since we asked it to!) and we assigned the first of those items to the variable a and the second to b. This technique is called *multiple assignment*.

Now let's assume that we've already written a function called `inverse(e, m)` that returns the multiplicative inverse of e modulo m—that is, the unique number $d < m$ such that $ed \mod m = 1$. We'll write that function in a moment, but assuming it exists, we have the ingredients for the `makeEncoderDecoder` function:

```
def makeEncoderDecoder():
    '''Returns two functions:  An RSA encryption function
       and an RSA decryption function.'''
    #
    # Choose 2 primes:
    #
    p, q = random.sample(primeSieve(range(2, 10)), 2)
    n = p*q           # compute n
    m = (p-1)*(q-1)   # compute m
    print ("Maximum number that can be encrypted is ", n-1)

    #
    # Choose a random prime for e:
    #
    e = random.choice(primeSieve(range(2, m)))
    if m % e == 0:    # If e divides m, it won't work!
        print ("Please try again")
        return
    else:
        d = inverse(e, m)               # compute d
```

```
    encoder = lambda x: (x**e) % n  # encryption function
    decoder = lambda y: (y**d) % n  # decryption function
    return [encoder, decoder]
```

Notice that this function is returning a list of two functions: the encryption and decryption functions! The encryption and decryption keys e and d are actual numbers that are embedded in those two functions, but x and y are the names of variables—the arguments that the user will ask to have encrypted or decrypted.

Finally, we need the `inverse(e, m)` function. We can implement it very simply by using `filter`, which we saw in Section 3.4. We're looking for the single number d such that $d < m$ and $ed \mod m = 1$. So we can generate all of the integers between 1 and $m - 1$ (which is done with `range(1, m)`) and then filter all of the values d such that $ed \mod m == 1$. Mathematicians tell us that there will be exactly one value of those. The `filter` function returns a list, so we use a sneaky Python trick and treat the call as if it were itself a list: we put `[0]` after it to pull out the first element and return it.

```
def inverse(e, m):
    '''Returns the inverse of e mod m'''
    return filter(lambda d: e*d % m == 1, range(1, m))[0]
```

You have to admit that this is amazing! However, *we* must admit that there are a few places where this function could be improved and extended. First, the prime numbers that we're choosing from for p and q are in the range from 2 to 10. We could easily change the 10 to something much larger, but we almost certainly don't want primes as small as 2. On the other hand, the `primeSieve` function expects to get a list of numbers that begins with 2 (do you see why?). To limit it to large prime numbers, we'd first need to generate primes beginning from 2 and then slice off the small ones. In addition, currently the `makeEncoderDecoder` might choose an encryption key e that is a divisor of m. In this case, we're told to try running the function again. That's not a very nice solution. You might try to think of ways to fix this so that `makeEncoderDecoder` would keep choosing values of e until it finds one that is not a divisor of m. You can do this with recursion, or you can do it with something called a `while` loop which we'll see in Chapter 5.

You might also rightfully object that encrypting numbers is not as interesting as encrypting strings (i.e., text). We'd agree. The good news is that strings can be converted into numbers and vice versa, as we'll see in the next chapter. Thus, it would not be difficult to encrypt strings by first converting them to numbers and then encrypting those numbers as we've done here. Decryption would first decrypt the number and then convert

that back into the original string. In fact, that's *exactly* how real encryption programs work!

3.9: Conclusion

I've been wondering what's going on inside my laptop.

In this chapter we've seen a beautiful idea: Functions are "first-class citizens" of the language, allowing us to treat them just like any other kind of data. In particular, functions can be the arguments and results of other functions. As a result, it's possible to have general-purpose higher-order functions like `map`, `reduce`, and `filter` that can be used in many different ways by simply providing them with appropriate functions of their own. Moreover, we can write other higher-order functions that produce functions as results, allowing us to easily write programs that write other programs.

Now that we have explored the foundations of programming, we will turn our attention to a new question: "How exactly does a computer actually manage to do all of this amazing stuff?" To answer that question, in the next chapter we're going to open the hood of a computer and see just how it works.

Chapter 4: Computer Organization

> *Computers are useless. They can only give you answers.*
>
> —Pablo Picasso

4.1 Introduction to Computer Organization

Hey!

When we run a Python program, what's actually going on inside the computer? While we hope that recursion is feeling less like magic to you now, the fact that an electronic device can actually interpret and execute something as complicated as a recursive program may seem - what's the right word here? - *alien.* The goal of this chapter is to metaphorically pry the lid off a computer and peer inside to understand what's really going on there.

As you can imagine, a computer is a complicated thing. A modern computer has on the order of billions of transistors. It would be impossible to keep track of how all of those components interact. Consequently, computer scientists and engineers design and think about computers using what are called *multiple levels of abstraction* . At the lowest level are components like transistors - the fundamental building blocks of modern electronic devices. Using transistors, we can build higher level devices called *logic gates* - they are the next level of abstraction. From logic gates we can build electronic devices that add, multiply, and do other basic operations - that's yet another level of abstraction. We keep moving up levels of abstraction, building more complex devices from more basic ones.

As a result, a computer can be designed by multiple people, each thinking about their specific level of abstraction. One type of expert might work on designing smaller, faster, and more efficient transistors. Another might work on using those transistors - never mind precisely how they work - to design better components that are based on transistors. Yet another expert will work on deciding how to organize these components into even more complex units that perform key computational functions. Each expert is grateful to be standing (metaphorically) on the shoulders of another person's work at the next lower level of abstraction.

By analogy, a builder thinks about building walls out of wood, nails, and sheet rock. An architect designs houses using walls without worrying too much about how they are built.

A city planner thinks about designing cities out of houses, without thinking too much how they are built, and so forth. This idea is called “abstraction” because it allows us to think about a particular level of design using the lower level ideas abstractly; that is, without having to keep in mind all of the specific details of what happens at that lower level.

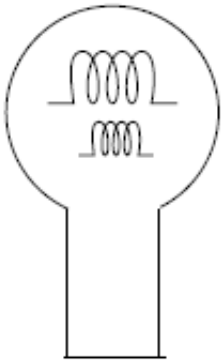


Figure 4.1: A three-way light bulb.

This seemingly simple idea of abstraction is one of the most important ideas in computer science. Not only are computers designed this way, but software is as well. Once we have basic workhorse functions like `map`, `reduce`, and others, we can use those to build more complex functions. We can use those more complex functions in many places to build even more complex software. In essence, we modularize so we that we can reuse good things to build bigger good things.

In this spirit, we’ll start by looking at how data is represented in a computer. Next, we’ll move up the levels of abstraction from transistors all the way up to a full-blown computer. We’ll program that computer in its own “native language” and talk about how your Python program ultimately gets translated to that language. By the end of this chapter, we’ll have a view of what happens when we run a program on our computer.

4.2 Representing Information

At the most fundamental level, a computer doesn’t really know math or have any notion of what it means to compute. Even when a machine adds 1+1 to get 2, it isn’t *really* dealing with numbers. Instead, it’s manipulating electricity according to specific rules.

To make those rules produce something that is useful to us, we need to associate the electrical signals inside the machine with the numbers and symbols that we, as humans, like to use.

4.2.1 Integers

That's a shocking idea!

The obvious way to relate electricity to numbers would be to assign a direct correspondence between voltage (or current) and numbers. For example, we could let zero volts represent the number 0, 1 volt be 1, 10 volts be 10, and so forth. There was a time when things were done this way, in so-called analog computers. But there are several problems with this approach, not the least of which would be the need to have a million-volt computer!

Whose bright idea was this?

Here's another approach. Imagine that we use a light bulb to represent numbers. If the bulb is off, the number is 0. If the bulb is on, the number is 1. That's fine, but it only allows us to represent two numbers.

All right then, let's upgrade to a "three-way" lamp. A three-way lamp really has four switch positions: off, and three increasing brightness levels. Internally, a three-way bulb has two filaments (Figure 4.1), one dim and one bright. For example, one might be 50 watts, and the other 100. By choosing neither of them, one, the other, or both, we can get 0, 50, 100, or 150 watts worth of light. We could use that bulb to represent the numbers 0, 50, 100, and 150 or we could decide that the four levels represent the numbers 0, 1, 2, and 3.

Internally, a computer uses this same idea to represent integers. Instead of using 50, 100, and 200, as in our illuminating example above, computers use combinations of numbers that are powers of 2. Let's imagine that we have a bulb with a 20-watt filament, a 21-watt filament, and a 22-watt filament. Then we could make the number 0 by turning none of the filaments on, we could make 1 by turning on only the 20-watt filament, we could make 2 by turning on the 21-watt filament, and so forth, up to $2^0 + 2^1 + 2^2 = 7$ using all three filaments.

Imagine now that we had the following four consecutive powers of 2 available to us: $2^0, 2^1, 2^2, 2^3$. Take a moment to try to write the numbers 0, 1, 2, and so forth as high as you can go by using zero or one of each of these powers of 2. Stop reading. We'll wait while you try this.

If all went well, you discovered that you could make all of the integers from 0 to 15 using 0 or 1 of each of these four powers of 2. For example, 13 can be represented as $2^0 + 2^2 + 2^3$. Written another way, this is:

$$13 = 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0$$

There are 10 kinds of people in this world: Those who understand binary and those who don't.

Notice that we've written the larger powers of 2 on the left and the smaller powers of two on the right. This convention is useful, as we'll see shortly. The 0's and 1's in the equation above - the *coefficients* on the powers of 2 - indicate whether or not the particular power of 2 is being used. These 0 and 1 coefficients are called *bits*, which stands for *bi*nary digi*ts*. *Binary* means "using two values"- here, the 0 and the 1 are the two values.

It's convenient to use the sequence of bits to represent a number, without explicitly showing the powers of two. For example, we would use the bit sequence 1101 to represent the number 13 since that is the order in which the bits appear in the equation above. Similarly 0011 represents the number 3. We often leave off the leading (that is, the leftmost) 0's, so we might just write this as 11 instead.

The representation that we've been using here is called base 2 because it is built on powers of 2. Are other bases possible? Sure! You use base 10 every day. In base 10, numbers are made out of powers of 10 and rather than just using 0 and 1 as the coefficients, we use 0 through 9. For example, the sequence 603 really means

$$6 \cdot 10^2 + 0 \cdot 10^1 + 3 \cdot 10^0$$

Other bases are also useful. For example, the Yuki Native American tribe, who lived in Northern California, used base 8. In base 8 we use powers of 8 and the coefficients that we use are 0 through 7. So, for example, the sequence 207 in base 8 means

$$2 \cdot 8^2 + 0 \cdot 8^1 + 7 \cdot 8^0$$

which is 135 (in base 10). It is believed that the Yukis used base 8 because they counted using the eight slots *between* their fingers.

In Star Wars, the Hutts have 8 fingers and therefore also count in base 8.

Notice that when we choose some base b (where b is some integer greater than or equal to 2), the digits that we use as coefficients are 0 through $b - 1$. Why? It's not hard to prove mathematically that when we use this convention, every positive integer between 0 and $bd - 1$ can be represented using d digits. Moreover, every integer in this range has a *unique* representation, which is handy since it avoids the headache of having multiple representations of the same number. For example, just as 42 has no other representation

in base 10, the number 1101 in base 2 (which we saw a moment ago is 13 in base 10) has no other representation in base 2.

Many older or smaller computers use 32 bits to represent a number in base 2; sensibly enough, we call them "32-bit computers." Therefore, we can uniquely represent all of the positive integers between 0 and 2^32 - 1 , or 4, 294, 967, 295. A powerful modern computer that uses 64 bits to represent each number can represent integers up to 2^64 - 1, which is roughly 18 *quadrillion*.

4.2.2 Arithmetic

We'll have sum fun with addition.

Arithmetic in base 2, base 8, or base 42 is analogous to arithmetic in base 10. For example, let's consider addition. In base 10, we simply start at the rightmost column, where we add those digits and "carry" to the next column if necessary. For example, when adding $17 + 25$, $5 + 7 = 12$ so we write down a 2 in the rightmost position and carry the 1. That 1 represents a 10 and is therefore propagated, or carried, to the next column, which represents the "10's place."

Although I find coming to agreement with myself to be easier in general.

Addition in base 2 is nearly identical. Let's add 111 (which, you'll recall is 7 in base 10) and 110 (which is 6 in base 10). We start at the rightmost (or "least significant") column and add 1 to 0, giving 1. Now we move to the next column, which is the 2^1 position or "2's place". Each of our two numbers has a 1 in this position. 1 + 1 = 2 but we only use 0's and 1's in base 2, so in base 2 we would have 1 + 1 = 10. This is analogous to adding 7 + 3 in base 10: rather than writing 10, we write a 0 and carry the 1 to the next column. Similarly, in base 2, for 1 + 1 we write 0 and carry the 1 to the next column. Do you see why this works? Having a 2 in the "2's place" is the same thing as having a 1 in the "4's place". In general having a 2 in the column corresponding to 2^i is the same as having a 1 in the next column, the one corresponding to 2^{i+1} since $2 \cdot 2^i = 2^{i+1}$.

Takeaway message: *Addition, subtraction, multiplication, and division in your favorite base are all analogous to those operations in base 10!*

4.2.3 Letters and Strings

We'll try not to get carried away with

As you know, computers don't only manipulate numbers; they also work with symbols, words, and documents. Now that we have ways to represent numbers as bits, we can use those numbers to represent other symbols.

these examples, but you should try adding a few numbers in base 2 to make sure that it makes sense to you.

It's fairly easy to represent the alphabet numerically; we just need to come to an agreement, or "convention," on the encoding. For example, we might decide that 1 should mean "A", 2 should mean "B", and so forth. Or we could represent "A" with 42 and "B" with 97; as long as we are working entirely within our own computer system it doesn't really matter.

*In ASCII, the number 42 represents the asterisk (*)*

But if we want to send a document to a friend, it helps to have an agreement with more than just ourselves. Long ago, the American National Standards Institute (ANSI) published such an agreement, called ASCII (pronounced "as key" and standing for the American Standard Code for Information Interchange). It defined encodings for the upper- and lower-case letters, the numbers, and a selected set of special characters - which, not coincidentally, happen to be precisely the symbols printed on the keys of a standard U.S. keyboard.

Negative Thinking

We've successfully represented numbers in base 2 and done arithmetic with them. But all of our numbers were positive. How about representing negative numbers? And what about fractions?

Let's start with negative numbers. One fairly obvious approach is to reserve one bit to indicate whether the number is positive or negative; for example, in a 32-bit computer we might use the leftmost bit for this purpose: Setting that bit to 0 could mean that the number represented by the remaining 31 bits is positive. If that leftmost bit is a 1 then the remaining number would be considered to be negative. This is called a *sign-magnitude* representation. The price we pay is that we lose half of our range (since we now have only 31 bits, in our example, to represent the magnitude of the number). While we don't mean to be too negative here, a bigger problem is that it's tricky to build computer circuits to manipulate sign-magnitude numbers. Instead, we use a system called two's complement.

The idea behind two's complement is this: It would be very convenient if the

representation of a number plus the representation of its negation added up to 0. For example, since 3 added to -3 is 0, it would be nice if the binary representation of 3 plus the binary representation of -3 added up to 0. We already know what the binary representation of 3 is 11. Let's imagine that we have an 8-bit computer (rather than 32- or 64-bit), just to make this example easier. Then, including the leading 0's, 3 would be represented as 00000011. Now, how could we represent -3 so that its representation added to 00000011 would be 0, that is 00000000?

Notice that if we "flip" all of the bits in the representation of 3, we get 11111100. Moreover, 00000011 + 11111100 = 11111111. If we add one more to this we get 11111111 + 00000001 and when we do the addition with carries we get 100000000; a 1 followed by eight 0's. If the computer is only using eight bits to represent each number then that leftmost ninth bit will not be recorded! So, what will be saved is just the lower eight bits, 00000000, which is 0. So, to represent -3, we can simply take the representation of 3, flip the bits, and then add 1 to that. (Try it out to make sure that you see how this works.) In general, the representation of a negative number in the two's complement system involves flipping the bits of the positive number and then adding 1.

You can look up the ASCII encoding on the web. Alternatively, you can use the Python function ord to find the numerical representation of any symbol. For example:

"ord" stands for "ordinal". You can think of this as asking for the "ordering number" of the symbol

```
>>> ord('*')
42
>>> ord('9')
57
```

Why is the ordinal value of '9' reported as 57? Keep in mind that the 9, in quotes, is just a character like the asterisk, a letter, or a punctuation symbol. It appears as character 57 in the ASCII convention. Incidentally, the inverse of ord is chr. Typing chr(42) will return an asterisk symbol and chr(57) will return the symbol '9'.

4 bits are sometimes called a "nybble"; we take no responsibility for this pathetically

Each character in ASCII can be represented by 8 bits, a chunk

commonly referred to as a "*byte*." Unfortunately, with only 8 bits ASCII can only represent 256 different symbols. (You might find it entertaining to pause here and write a short program that counts from 0 to 255 and, for each of these numbers, prints out the ASCII symbol corresponding to that number. You'll find that some of the symbols printed are "weird" or even invisible. Snoop on the Web to learn more about why this is so.)

nerdy pun.

Piecing It Together

How about representing fractions? One approach (often used in video and music players) is to establish a convention that everything is measured in units of some convenient fraction (just as our three-way bulb works in units of 50 watts). For example, we might decide that everything is in units of 0.01, so that the number 100111010 doesn't represent 314 but rather represents 3.14.

However, scientific computation often requires both more precision and a wider range of numbers than this strategy affords. For example, chemists often work with values as on the order of 10^{23} or more (Avogadro's number is approximately 6.02×10^{23}), while a nuclear physicist might use values as small as 10^{-12} or even smaller.

Imagine that we are operating in base 10 and we have only eight digits to represent our numbers. We might use the first six digits to represent a number, with the convention that there is an implicit 0 followed by a decimal point, just before the first digit. For example, the six digits 123456 would represent the number 0.123456. Then, the last two digits could be used to represent the exponent on the power of 10. So, 12345678 would represent 0.123456×10^{78} . Computers use a similar idea to represent fractional numbers, except that base 2 is used instead of base 10.

It may seem that 256 symbols is a lot, but it doesn't provide for accented characters used in languages like French (Français), let alone the Cyrillic or Sanskrit alphabets or the many thousands of symbols used in Chinese and Japanese.

To address that oversight, the International Standards Organization (ISO) eventually devised a system called Unicode, which can represent every character in every known language, with room for future growth. Because Unicode is somewhat

wasteful of space for English documents, ISO also defined several “Unicode Transformation Formats” (UTF), the most popular of which is *UTF-8*. You may already be using UTF-8 on your computer, but we won’t go into the gory details here.

There are even unofficial Unicode symbols for Klingon!

Of course, individual letters aren’t very interesting. Humans normally like to string letters together, and we’ve seen that Python uses a data type called “strings” to do this. It’s easy to do that with a sequence of numbers; for example, in ASCII the sequence 99, 104, 111, 99, 111, 108, 97, 116, 101 translates to “chocolate”. The only detail missing is that when you are given a long string of numbers, you have to know when to stop; a common convention is to include a “length field” at the very beginning of the sequence. This number tells us how many characters are in the string. (Python uses a length field, but hides it from us to keep the string from appearing cluttered.)

4.2.4 Structured Information

Using the same concepts, we can represent almost any information as a sequence of numbers. For example, a picture can be represented as a sequence of colored dots, arranged in rows. Each colored dot (also known as a “picture element” or pixel) can be represented as three numbers giving the amount of red, green, and blue at that pixel. Similarly, a sound is a time sequence of “sound pressure levels” in the air. A movie is a more complex time sequence of single pictures, usually 24 or 30 per second, along with a matching sound sequence.

That’s the level of abstraction thing again! Bits make up numbers, numbers make up pixels, pixels make up pictures, pictures make up movies. A two-hour movie can require several billion bits, but nobody who is making or watching a movie wants to think about all of those bits!

That would be more than a bit annoying!

4.3 Logic Circuitry

Now that we have adopted some conventions on the representation of data it’s time to build devices that manipulate data. We’ll start at a low level of abstraction of transistors and move up the metaphorical “food chain” to more complex devices, then units that can perform addition and other basic operations, and finally to a full-blown computer.

My favorite food chain sells donuts.

4.3.1 Boolean Algebra

In Chapter 2 we talked about Boolean variables - variables that take the value `True` or `False`. It turns out that Booleans are at the very heart of how a computer works.

As we noted in the last section, it's convenient to represent our data in base 2, also known as binary. The binary system has two digits, 0 and 1 just as Boolean variables have two values, `False` and `True`. In fact, we can think of 0 as corresponding to `False` and 1 as `True` as corresponding to `True`. The truth is, that Python thinks about it this way too. One funny way to see this is as follows:

```
>>> False + 42
42
>>> True + 2
3
```

Weird - but there it is: in Python `False` is really 0 and `True` is really 1. By the way, in many programming languages this is not the case. In fact, programming language designers have interesting debates about whether it's a good idea or not to have `False` and `True` be so directly associated with the numbers 0 and 1. On the one hand, that's how we often think about `False` and `True`. On the other hand, it can result in confusing expressions like `False + 42` which are hard to read and prone to introducing programmer errors.

We are writing these in upper-case letters to indicate that we are talking about operations on bits- 0's and 1's - rather than Python's build in `and`, `or`, *and* `not`.

With the Booleans `True` and `False` we saw that we could use the operations and, or, and not to build up more interesting Boolean expressions. For example, `True` and `True` is the same as `True` while `True` or `False` is `True` and not `True` is `False`. Of course, we can now emulate these three operations for 0 and 1. 1 `AND` 1 = 1, 1 `OR` 0 = 1, and `NOT` 1 = 0.

Although your intuition of `AND`, `OR`, and `NOT` is probably fine, we can be very precise about these three operations by defining them with a *truth table* : a listing of all possible combinations of values of the input variables, together with the result produced by the function. For example, the truth table for `AND` is:

x	y	x AND y
0	0	0
0	1	0
1	0	0
1	1	1

In Boolean notation, AND is normally represented as multiplication; an examination of the above table shows that as long as *x* and *y* are either 0 or 1, *x* AND *y* is in fact identical to multiplication. Therefore, we will often write *xy* to represent *x* AND *y*.

Takeaway message: AND *is 1 if and only if both of its arguments are 1.*

OR is a two-argument function that is 1 if either of its arguments are 1. The truth table for OR is:

x	y	x OR y
0	0	0
0	1	1
1	0	1
1	1	1

OR is normally written using the plus sign: $x + y$. The first three lines of the above table are indeed identical to addition, but note that the fourth is different.

Takeaway message: OR *is 1 if either of its arguments is 1.*

Finally, NOT is a one-argument function that produces the opposite of its argument. The truth table is:

x	NOT x
0	1
1	0

NOT is normally written using an overbar, e.g. $\bar{x}$

4.3.2 Making Other Boolean Functions

Amazingly, any function of Boolean variables, no matter how complex, can be expressed in terms of AND, OR, and NOT. No other operations are required because, as we'll see, any other operation could be made out of AND, OR, and NOT s. In this section we'll show how to do that. This fundamental result will allow us to build circuits to do things like arithmetic

and, ultimately, a computer. For example, consider a function described by the truth table below. This function is known as "implication" and is written $x \Rightarrow y$.

x	y	$x \Rightarrow y$
0	0	1
0	1	1
1	0	0
1	1	1

This function can be expressed as $\bar{x} + xy$. To see why, try building the truth table for $\bar{x} + xy$. That is, for each of the four possible combinations of *x* and *y*, evaluate $\bar{x} + xy$. For example, when $x = 0$ and $y = 0$, notice that $\bar{x}$ is 1. Since the OR of 1 and anything else is always 1, we see that $\bar{x} + xy$ evaluates to 1 in this case. Aha! This is exactly the value that we got in the truth table above for $x = 0$ and $y = 0$. If you continue doing this for the next three rows, you'll see that values of $x \Rightarrow y$ and $\bar{x} + xy$ always match. In other words, they are identical. This method of enumerating the output for each possible input is a fool-proof way of proving that two functions are identical, even if it is a bit laborious.

For simple Boolean functions, it's often possible to invent an expression for the function by just inspecting the truth table. However, it's not always so easy to do this, particularly when we have Boolean functions with more than two inputs. So, it would be nice to have a systematic approach for building expressions from truth tables. The *minterm expansion principle* provides us with just such an approach.

Perhaps "minterms" should be called "happyterms".

We'll see how the minterm expansion principle works through an example. Specifically, let's try it out for the truth table for the implication function above. Notice that when the inputs are $x = 1$ and $y = 0$, the truth table tells us that the output is 0. However, for all three of the other rows (that is pairs of inputs), the output is more "interesting" - it's 1. We'll build a custom-made logical expression for each of these rows with a 1 as the output. First, consider the row $x = 0 ; y = 0$. Note that the expression $\bar{x}\bar{y}$ evaluates to 1 for this particular pair of inputs because the NOT of 0 is 1 and the AND of 1 and 1 is 1. Moreover, notice that for every other possible pair of values for x and y this term $\bar{x}\bar{y}$ evaluates to 0. Do you see why? The only way that $\bar{x}\bar{y}$ can evaluate to 1 is for $\bar{x}$ to be 1 (and thus for x to be 0) and for $\bar{y}$ to be 1 (since we are computing the AND here and AND outputs 1 only if both of its inputs are 1). The term $\bar{x}\bar{y}$ is called a minterm. You can think of it as being custom-made to make the inputs $x = 0 ; y = 0$ "happy" (evaluate to 1) and does nothing for every other pair of inputs.

We're not done yet! We now want a minterm that is custom-made to evaluate to 1 for the input $x = 0; y = 1$ and evaluates to 0 for every other pair of input values. Take a moment to try to write such a minterm. It's $\bar{x}y$. This term evaluates to 1 if and only if $x = 0$ and $y = 1$. Similarly, a minterm for $x = 1; y = 1$ is math:*xy*. Now that we have these minterms, one for each row in the truth table that contains a 1 as output, what do we do next? Notice that in our example, our function should output a 1 if the first minterm evaluates to 1 or the second minterm evaluates to 1 or the third minterm evaluates to 1. Also notice the words "or" in that sentence. We want to OR the values of these three minterms together. This gives us the expression $\bar{x}\bar{y} + \bar{x}y + xy$. This expression evaluates to 1 for the first, second, and fourth rows of the truth table as it should. How about the third row, the "uninteresting" case where $x = 1; y = 1$ should output 0. Recall that each of the minterms in our expression was custom-made to make exactly one pattern "happy". So, none of these terms will make the $x = 1; y = 1$ "happy" and thus, for that pair of inputs, our newly minted expression outputs a 0 as it should!

It's not hard to see that this minterm expansion principle works for every truth table. Here is the precise description of the process:

1. Write down the truth table for the Boolean function that you are considering.
2. Delete all rows from the truth table where the value of the function is 0.
3. For each remaining row we will create something called a "minterm" as follows:
    a. For each variable that has a 1 in that row, write the name of the variable. If the input variable is 0 in that row, write the variable with a negation symbol to NOT it.
    b. Now AND all of these variables together.
4. Combine all of the minterms for the rows using OR .

You might have noticed that this general algorithm for converting truth tables to logic expressions only uses AND, OR, and NOT operations. It uses NOT and AND to construct each minterm and then it uses OR to "glue" these minterms together. This effectively proves that AND, OR, and NOT suffice to represent any Boolean function!

The minterm expansion principle is a recipe - it's an *algorithm*. In fact, it can be implemented on a computer to automatically construct a logical expression for any truth table. In practice, this process is generally done by computers. Notice, however that this algorithm doesn't necessarily give us the simplest expression possible. For example, for

the implication function, we saw that the expression $\bar{x} + xy$ is correct. However, the minterm expansion principle produced the expression $\bar{x}\bar{y} + \bar{x}y + xy$. These expressions are logically equivalent, but the first one is undeniably shorter. Regrettably, the so-called "minimum equivalent expressions" problem of finding the shortest expression for a Boolean function is very hard. In fact, a Harvey Mudd College graduate, David Buchfuhrer, recently showed that the minimum equivalent expressions problem is provably as hard as some of the hardest (unsolved) problems in mathematics and computer science. Amazing but true!

4.3.3 Logic Using Electrical Circuits

That means that computers are helping design other computers! That seems profoundly amazing to me.

Next, we'd like to be able to perform Boolean functions in hardware. Let's imagine that our basic "building block" is an electromagnetic switch as shown in Figure 4.2. There is *always* power supplied to the switch (as shown in the upper left) and there is a spring that holds a movable "arm" in the up position. So, normally, there is no power going to the wire labeled "output". The "user's" input is indicated by the wire labeled "input." When the input power is off (or "low"), the electromagnet is not activated and the movable arm remains up, and output is 0. When the input is on (or "high"), the electromagnet is activated, causing the movable arm to swing downwards and power to flow to the output wire. Let's agree that a "low" electrical signal corresponds to the number 0 and a "high" electrical signal corresponds to the number 1. Now, let's build a device for computing the AND function using switches. We can do this as shown in Figure 4.3 where there are two switches in series. In this figure, we use a simple representation of the switch without the power or the ground. The inputs are x and y, so when x is 1, the arm of the first switch swings down and closes the switch, allowing power to flow from the left to the right. Similarly, when y is 1, that arm of the second switch swings down, allowing power to flow from left to right. Notice that when either or both of the input x, y are 0, at least one switch remains open and there is no electrical signal flowing from the power source to the output. However, when both x *and* y are 1 both switches close and there is a signal, that is a 1, flowing to the output. This is a device for computing x AND y. We call this an AND gate.

Similarly, the circuit in Figure 4.4 computes x OR y and is called an OR gate. The function NOT x can be implemented by constructing a switch that conducts if and only x is 0.

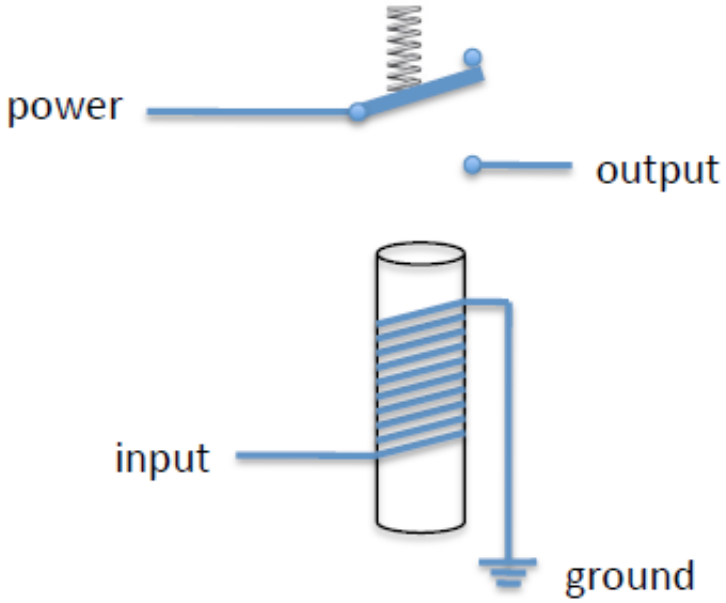


Figure 4.2: An electromagnetic switch.

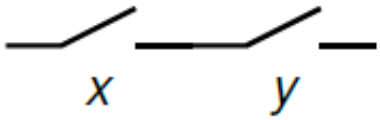


Figure 4.3: An AND gate constructed with switches.

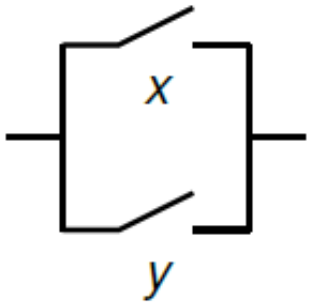


Figure 4.4: An OR gate constructed with switches.

While computers based on electromechanical switches were state-of-the-art in the 1930's, computers today are built with transistorized switches that use the same principles but are much smaller, much faster, more reliable, and more efficient. Since the details of the switches aren't terribly important at this level of

Those gate shapes are weird. I'm on the

abstraction, we represent, or "abstract", the gates using symbols as shown in Figure 4.5

fence about whether I like them or not.

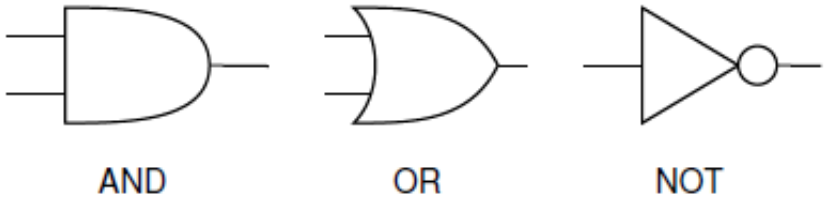


Figure 4.5: Symbols used for AND, OR, and NOT gates.

We can now build circuits for any Boolean function! Starting with the truth table, we use the minterm expansion principle to find an expression for the function. For example, we used the minterm expansion principle to construct the expression $\bar{x}\bar{y} + \bar{x}y + xy$ for the implication function. We can convert this into a circuit using AND, OR, and NOT gates as shown in Figure 4.6.

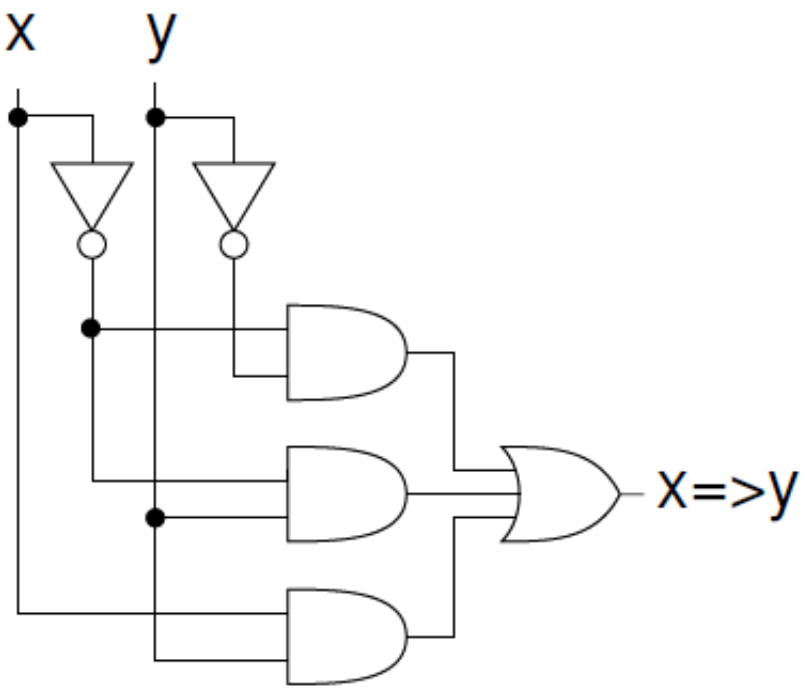


Figure 4.6: A circuit for the implication function.

4.3.4 Computing With Logic

Now that we know how to implement functions of Boolean variables, let's move up one more level of abstraction and try to build some units that do arithmetic. In binary, the numbers from 0 to 3 are represented as 00, 01, 10, and 11. We can use a simple truth

table to describe how to add two two-bit numbers to get a three-bit result:

x	y	$x+y$
00	00	000
00	01	001
00	10	010
⋮		⋮
01	10	011
01	11	100
⋮		⋮
11	11	110

In all, this truth table contains sixteen rows. But how can we apply the minterm expansion principle to it? The trick is to view it as three tables, one for each bit of the output. We can write down the table for the rightmost output bit separately, and then create a circuit to compute that output bit. Next, we can repeat this process for the middle output bit. Finally, we can do this one more time for the leftmost output bit. While this works, it is much more complicated than we would like! If we use this technique to add two 16-bit numbers, there will be 2^{32} rows in our truth table resulting in several *billion* gates.

Ouch!

Fortunately, there is a much better way of doing business. Remember that two numbers are added (in any base) by first adding the digits in the rightmost column. Then, we add the digits in the next column and so forth, proceeding from one column to the next, until we're done. Of course, we may also need to carry a digit from one column to the next as we add.

We can exploit this addition algorithm by building a relatively simple circuit that does just one column of addition. Such a device is called a *full adder*, (admittedly a funny name given that it's only doing one column of addition!). Then we can "chain" 16 full adders together to add two 16-bit numbers or chain 64 copies of this device together if we want to add two 64-bit numbers. The resulting circuit, called a *ripple-carry adder*, will be much simpler and smaller than the first approach that we suggested above. This modular approach allows us to first design a component of intermediate complexity (e.g. the full adder) and use that design to design a more complex device (e.g. a 16-bit adder). Aha! Abstraction again!

The full adder takes three inputs: The two digits being added in this column (we'll call them x and y) and the "carry in" value that was propagated from the previous column (we'll call that c_{in}). There will be two outputs: The sum (we'll call that z) and the "carry

out" to be propagated to the next column (we'll call that c_{out}). We suggest that you pause here and build the truth table for this function. Since there are three inputs, there will be $2^3 = 8$ rows in the truth table. There will be two columns of output. Now, treat each of these two output columns as a separate function. Starting with the first column of output, the sum z, use the minterm expansion principle to write a logic expression for z. Then, convert this expression into a circuit using AND, OR, and NOT gates. Then, repeat this process for the second column of output, c_{out}. Now you have a full adder! Count the gates in this adder - it's not a very big number.

Finally, we can represent this full adder abstractly with a box that has the three inputs on top and the two outputs on the bottom. We now chain these together to build our ripple-carry adder. A 2-bit ripple-carry adder is shown in Figure 4.7. How many gates would be used in total for a 16-bit ripple-carry adder? It's in the hundreds rather than the billions required in our first approach!

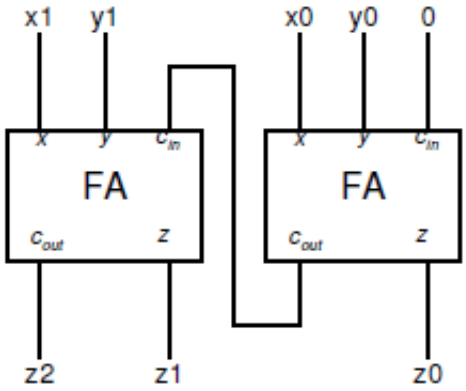


Figure 4.7: A 2-bit ripple-carry adder. Each box labeled "FA" is a full adder that accepts two input bits plus a carry, and produces a single output bit along with a carry into the next FA.

Now that we've built a ripple-carry adder, it's not a big stretch to build up many of the other kinds of basic features of a computer. For example, consider building a multiplication circuit. We can observe that multiplication involves a number of addition steps. Now that we have an addition module, that is an abstraction that we can use to build a multiplier!

Take Away *Using the minterm expansion principle and modular design, we can now build virtually all of the major parts of a computer.*

4.3.5 Memory

There is one important aspect of a computer that we haven't seen how to design: memory! A computer can store data and then fetch those data for later use.("Data" is the plural of "datum". Therefore, we say "those data" rather than "that data".) In this section we'll see how to build a circuit that stores a single bit (a 0 or 1). This device is called a *latch* because it allows us to "lock" a bit and retrieve it later. Once we've built a latch, we can abstract that into a "black box" and use the principle of modular design to assemble many latches into a device that stores a lot of data.

I'm glad that you didn't forget this part!

A latch can be created from two interconnected NOR gates: NOR is just OR followed by NOT. That is, its truth table is exactly the opposite of the truth table for OR as shown below.

x	y	x NOR y
0	0	1
0	1	0
1	0	0
1	1	0

A NOR gate is represented symbolically as an OR gate with a little circle at its output (representing negation).

Now, a latch can be constructed from two NOR gates as shown in Figure 4.8. The input S is known as "set" while the input R is known as "reset". The appropriateness of these names will become evident in a moment.

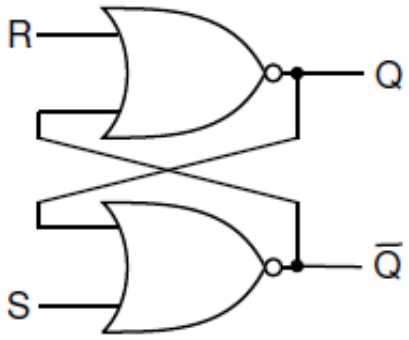


Figure 4.8: A latch built from two NOR gates.

What in the world is going on in this circuit!? To begin with, suppose that all of R, S, and Q are 0. Since both Q and S are 0, the output of the bottom NOR gate, $\bar{Q}$, is 1. But since $\bar{Q}$ is 1, the top NOR gate is forced to produce a 0. Thus, the latch is in a stable state: the fact that $\bar{Q}$ is 1 holds Q at 0.

Now, consider what happens if we change S (remember, it's called "set") to 1, while holding R at 0. This change forces $\bar{Q}$ to 0; for a moment, both Q and $\bar{Q}$ are zero. But the fact that $\bar{Q}$ is zero means that both inputs to the top NOR gate are zero, so its output, Q, must become 1. After that happens, we can return S to 0, and the latch will remain stable. We can think of the effect of changing S to 1 for a moment as "setting" the latch to store the value 1. The value is stored at Q. ($\bar{Q}$ is just used to make this circuit do its job, but it's the value of Q that we will be interested in.)

An identical argument will show that R (remember, it's called "reset") will cause Q to return to zero, and thus $\bar{Q}$ will become 1 again. That is, the value of Q is reset to 0! This circuit is commonly called the *S-R latch*.

What happens if both S and R become 1 at the same time? Try this out through a thought experiment. We'll pause here and wait for you.

Did you notice how this latch will misbehave? When S and R are both set to 1 (this is trying to set and reset the circuit simultaneously—very naughty) both Q and $\bar{Q}$ will become 0. Now, if we let S and R return back to 0, the inputs to both NOR gates are 0 and their outputs both become 1. Now each NOR gate gets a 1 back as input and its output becomes 0. In other words, the NOR gates are rapidly "flickering" between 0 and 1 and not storing anything! In fact, other weird and unpredictable things can happen if the two NOR gates compute their outputs at slightly different speeds. Circuit designers have found ways to avoid this problem by building some "protective" circuitry that ensures that S and R can never be simultaneously set to 1.

So, a latch is a one-bit memory. If you want to remember a 1, turn S to 1 for a moment; if you want to remember a 0, turn R to 1 for a moment. If you aggregate 8 latches together, you can remember an 8-bit byte. If you aggregate millions of bits, organized in groups of 8, you have the *Random Access Memory (RAM)* that forms the memory of a computer.

I'm sheepish about sharing my RAM puns because you've probably herd them already.

4.4 Building a Complete Computer

Imagine that you've been elected treasurer of your school's unicycling club and its time to do the books and balance the budget. You have a large notebook with all of the club's finances. As you work, you copy a few numbers from the binder onto a

"Balancing" the club's budget?! We promise that wheel have no more

scratch sheet, do some computations using a calculator, and jot those results down on your scratch sheet. Occasionally, you might copy some of those results back into the big notebook to save for the future and then jot some more numbers from the notebook onto your scratch sheet.

unicycle jokes!

A modern computer operates on the same principle. Your calculator and the scratch sheet correspond to the *CPU* of the computer. The CPU is where computation is performed but there's not enough memory there to store all of the data that you will need. The big notebook corresponds to the computer's memory. These two parts of the computer are physically separate components that are connected by wires on your computer's circuit board.

We spoke too soon about no more unicycle jokes.

What's in the CPU? There are devices like ripple-carry adders, multipliers, and their ilk for doing arithmetic. These devices can all be built using the concepts that we saw earlier in this chapter, namely the minterm expansion principle and modular design. The CPU also has a small amount of memory, corresponding to the scratch sheet. This memory comprises a small number of *registers* where we can store data. These registers could be built out of latches or other related devices. Computers typically have on the order of 16 to 32 of these registers, each of which can store 32 or 64 bits. All of the CPU's arithmetic is done using values in the registers. That is, the adders, multipliers, and so forth expect to get their inputs from registers and to save the results to a register, just as you would expect to use your scratch pad for the input and output of your computations.

The memory, corresponding to the big notebook, can store a lot of data - probably billions of bits! When the CPU needs data that is not currently stored in a register (scratch pad), it requests that data from memory. Similarly, when the CPU needs to store the contents of a register (perhaps because it needs to use that register to store some other values), it can ship it off to be stored in memory.

See! We didn't tire you with any more unicycle puns.

What's the point of having separate registers and memory? Why not just have all the memory in the CPU? The answer is multifaceted, but here is part of it: The CPU needs to be small in order to be fast. Transmitting a bit of data along a millimeter of wire slows the computer down considerably! On the other hand, the memory needs to be large in order to store lots of data. Putting a large memory in the CPU would make the CPU slow. In addition, there are other considerations that necessitate separating the CPU from the memory. For example, CPUs are built using different (and generally much more expensive) manufacturing processes than memories.

Now, let's complicate the picture slightly. Imagine that the process of balancing the unicycle club's budget is complicated. The steps required to do the finances involve making decisions along the way (e.g. "If we spent more than $500 on unicycle seats this year, we are eligible for a rebate.") and other complications. So, there is a long sequence of instructions that is written in the first few pages of our club notebook. This set of instructions is a program! Since the program is too long and complicated for you to remember, you copy the instructions one-by-one from the notebook onto your scratch sheet. You follow that instruction, which might tell you, for example, to add some numbers and store them some place. Then, you fetch the next instruction from the notebook.

Memory is slow. If the CPU can read from or write to a register in one unit of time, it will take approximately 100 units of time to read from or write to memory!

How do you remember which instruction to fetch next and how do you remember the instruction itself? The CPU of a computer has two special registers just for this purpose. A register called the *program counter* keeps track of the location in memory where it will find the next instruction. That instruction is then fetched from memory and stored in a special register called the *instruction register*. The computer examines the contents of the instruction register, executes that instruction, and then increments the program counter so that it will now fetch the next instruction.

John von Neumann (1903-1957)

One of the great pioneers of computing was John von Neumann(pronounced "NOY-mahn"), a Hungarian-born mathematician who contributed to fields as diverse as set theory and nuclear physics. He invented the Monte Carlo method (which we used to calculate π in Section 1.1.2), cellular automata, the *merge sort* method for sorting, and of course the von Neumann architecture for computers.

Although von Neumann was famous for wearing a three-piece suit everywhere—including on a mule trip in the Grand Canyon and even on the tennis court—he was not a boring person. His parties were always popular (although he sometimes sneaked away from his guests to work) and he loved to quote from his voluminous memory of off-color limericks. Despite his brilliance, he was a notoriously bad driver, which might explain why he bought a new car every year.

Von Neumann died of cancer, perhaps caused by radiation from atomic-bomb

tests. But his legacy lives on in every computer built today.

This way of organizing computation was invented by the famous mathematician and physicist, Dr. John von Neumann, and is known as the *von Neumann architecture*. While computers differ in all kinds of ways, they all use this fundamental principle. In the next subsection we'll look more closely at how this principle is used in a real computer.

4.4.1 The von Neumann Architecture

One of von Neumann's colleagues was Dr. Claude Shannon—inventor of the minterm expansion principle. Shannon, it turns out, was also a very good unicyclist.

We mentioned that the computer's memory stores both instructions and data. We know that data can be encoded as numbers, and numbers can be encoded in binary. But what about the instructions? Good news! Instructions too can be stored as numbers by simply adopting some convention that maps instructions to numbers.

For example, let's assume that our computer is based on 8-bit numbers and let's assume that our computer only has four instructions: add, subtract, multiply, and divide. (That's very few instructions, but let's start there for now and then expand later). Each of these instructions will need a number, called an *operation code* (or *opcode*), to represent it. Since there are four opcodes, we'll need four numbers, which means two bits per number. For example, we might choose the following opcodes for our instructions:

Operation	Meaning
00	Add
01	Subtract
10	Multiply
11	Divide

Next, let's assume that our computer has four registers number 0 through 3. Imagine that we want to add two numbers. We must specify the two registers whose values we wish to add and the register where we wish to store the result. If we want to add the contents of register 2 with the contents of register 0 and store the result in register 3, we could adopt the convention that we'll write "add 3, 0, 2". The last two numbers are the registers where we'll get our inputs and the first number is the register where we'll store our result. In binary, "add 3, 0, 2" would be represented as "00 11 00 10". We've added the spaces to help you see the numbers 00 (indicating "add"), 11 (indicating the register where the

result will be stored), 00 (indicating register 0 as the first register that we will add), and 10 (indicating register 2 as the second register that we will add).

Computer scientists often start counting from 0.

In general, we can establish the convention that an instruction will be encoded using 8 bits as follows: The first two bits (which we'll call I0 and I1) represent the instruction, the next two bits (D0 and D1) encode the "destination register" where our result will be stored, the next two bits (S0 and S1) encode the first register that we will add, and the last two bits (T0 and T1) encode the second register that we will add. This representation is shown below.

I0 I1	D0 D1	S0 S1	T0 T1

Recall that we assume that our computer is based on 8-bit numbers. That is, each register stores 8 bits and each number in memory is 8 bits long. Figure 4.9 shows what our computer might look like. Notice the program counter at the top of the CPU. Recall that this register contains a number that tells us where in memory to fetch the next instruction. At the moment, this program counter is 00000000, indicating address 0 in memory.

So a computer is kind of like a dog?

The computer begins by going to this memory location and fetching the data that resides there. Now look at the memory, shown on the right side of the figure. The memory addresses are given both in binary (base 2) and in base 10. Memory location 0 contains the data 00100010. This 8-bit sequence is now brought into the CPU and stored in the instruction register. The CPU's logic gates decode this instruction. The leading 00 indicates it's an addition instruction. The following 10 indicates that the result of the addition that we're about to perform will be stored in register 2. The next 00 and 10 mean that we'll get the data to add from registers 0 and 2, respectively. These values are then sent to the CPU's ripple-carry adder where they are added. Since registers 0 and 2 contain 00000101 and 00001010, respectively, before the operation, after the operation register 2 will contain the value 00001111.

In general, our computer operates by repeatedly performing the following procedure:

1. Send the address in the program counter (commonly called the *PC*) to the memory, asking it to read that location.
2. Load the value from memory into the instruction register.

3. *Decode* the instruction register to determine what instruction to execute and which registers to use.
4. *Execute* the requested instruction. This step often involves reading operands from registers, performing arithmetic, and sending the results back to the destination register. Doing so usually involves several sub-steps.
5. Increment the PC (Program Counter) so that it contains the address of the next instruction in memory. (It is this step that gives the PC its name, because it *counts* its way through the addresses in the program.)

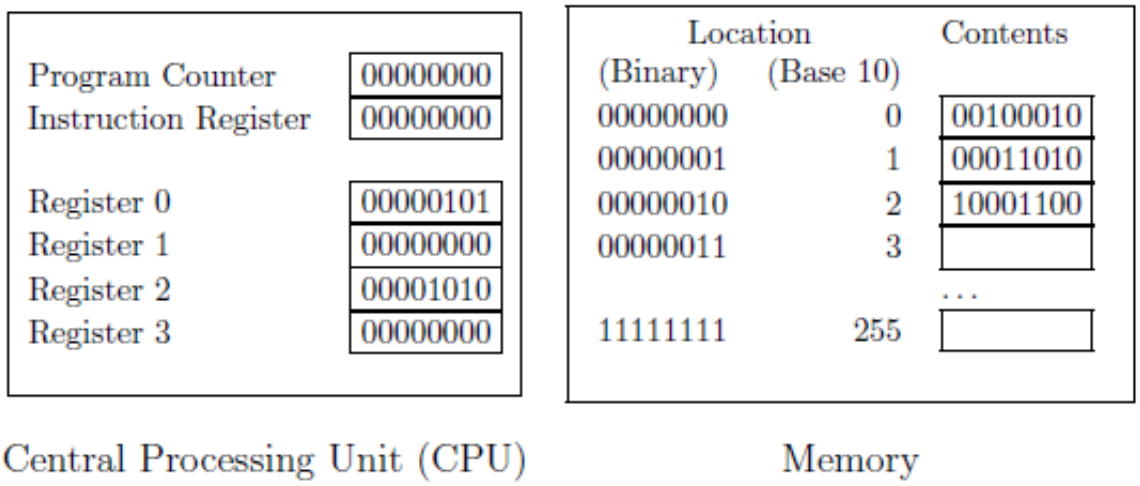


Figure 4.9: A computer with instructions stored in memory. The program counter tells the computer where to get the next instruction.

"Wait!" we hear you scream. "The memory is storing *both* instructions *and* data! How can it tell which is which?!" That's a great question and we're glad you asked. The truth is that the computer *can't tell* which is which. If we're not careful, the computer might fetch something from memory into its instruction register and try to execute it when, in fact, that 8-bit number represents the number of pizzas that the unicycle club purchased and not an instruction! One way to deal with this is to have an additional special instruction called "halt" that tells the computer to stop fetching instructions. In the next subsections we'll expand our computer to have more instructions (including "halt") and more registers and we'll write some real programs in the language of the computer.

Takeaway message: *A computer uses a simple process of repeatedly fetching its next instruction from memory, decoding that instruction, executing that instruction, and incrementing the program counter. All of these steps are implemented with digital circuits that, ultimately, can be built using the processes that we've described earlier in this chapter.*

4.5 Hmmm

The computer we discussed in Section 4.4 is simple to understand, but its simplicity means it's not very useful. In particular, a real computer also needs (at a minimum) ways to:

1. Move information between the registers and a large memory,
2. Get data from the outside world,
3. Print results, and
4. Make decisions.

To illustrate how these features can be included, we have designed the Harvey Mudd Miniature Machine, or Hmmm. Just like our earlier 4-instruction computer, Hmmm has a program counter, an instruction register, a set of data registers, and a memory. These are organized as follows:

1. While our simple computer used 8-bit instructions, in Hmmm, both instructions and data are 16 bits wide. The set of bits representing an instruction is called a *word*. That allows us to represent a reasonable range of numbers, and lets the instructions be more complicated.
2. In addition to the program counter and instruction register, Hmmm has 16 registers, named R0 through R15. R0 is special: it always contains zero, and anything you try to store there is thrown away.
3. Hmmm's memory contains 256 locations. Although the memory can be used for either instructions or data, programs are prevented from reading or writing the instruction section of memory. (Some modern computers offer a similar feature.)

Hmmm is music to my ears!

Different computers have different word sizes. Most machines sold today use 64-bit words; older ones use 32 bits. 16-bit, 8-bit, and even 4-bit computers are still used for special applications. Your digital watch probably contains a 4-bit computer.

It's very inconvenient to program Hmmm by writing down the bits corresponding to the instructions. Instead, we will use *assembly language*, which is a programming language where each machine instruction receives a friendlier symbolic representation. For example, to compute R3 = R1 + R2, we would write:

```
add r3, r1, r2
```

Hmmm's instruction

A very simple process is used to convert this assembly language into the 0's and 1's - the "machine language" - that the computer can execute.

set is large, but not hmmmungous.

A complete list of Hmmm instructions, including their binary encoding, is given in Figure 4.10.

4.5.1 A Simple Hmmm Program

To begin with, let's look at a program that will calculate the approximate area of a triangle. It's admittedly mundane, but it will help us move on to more interesing Hmmm-ing shortly. (We suggest that you follow along by downloading Hmmm from

Not the least of which is that writing even a short program in assembly language can be a hmmmbling experience!

http://www.cs.hmc.edu/~cs5grad/cs5/hmmm/documentation/documentation.html and trying these examples out with us.)

Let's begin by using our favorite editor to create a file named `triangle1.hmmm` with the following contents:

```
#
# Calculate the approximate area of a triangle.
#
# First input: base
# Second input: height
# Output: area
#

0       read    r1       # Get base
1       read    r2       # Get height
2       mul     r1 r1 r2 # b times h into r1
3       setn    r2 2
4       div     r1 r1 r2 # Divide by 2
5       write   r1
6       halt
```

How Does It Work?

What does all of this mean? First, anything starting with a pound sign ("#") is a comment; Hmmm ignores it. Second, you'll note that each line is numbered, starting with zero. This number indicates the memory location where the instruction will be stored.

You may have also noticed that this program doesn't use any commas, unlike the example `add` instruction above. Hmmm is very lenient about notation; all of the following instructions mean exactly the same thing:

```
add r1,r2,r3

ADD R1 R2 R3

ADD R1,r2, R3

aDd R1,,R2,       ,R3
```

Needless to say, we don't recommend the last two options!

So what do all of these lines actually do? The first two (0 and 1) read in the base and height of the triangle. When Hmmm executes a `read` instruction, it pauses and prompts the user to enter a number, converts the user's number into binary, and stores it into the named register. So the first-typed number will go into register R1, and the second into R2.

The `MUL`tiply instruction then finds $b \times h$ by calculating R1 = R1 $\times$ R2. This instruction illustrates three important principles of hmmm programming:

1. Most arithmetic instructions accept three registers: two *sources* and a *destination*.
2. The destination register is always listed first, so that the instruction can be read somewhat like a Python assignment statement.
3. A source and destination can be the same.

After multiplying, we need to divide $b \times h$ by 2. But where can we get the constant 2? One option would be to ask the user to provide it via a `read` instruction, but that seems clumsy. Instead, a special instruction, `setn` (*set to number*), lets us insert a small constant into a register. As with `mul`, the destination is given first; this is equivalent to the Python statement R2 = 2.

The `DIV`ide instruction finishes the calculation, and `write` displays the result on the screen. There is one thing left to do, though: after the `write` is finished, the computer will happily try to execute the instruction at the following memory location. Since there isn't a

valid instruction there, the computer will fetch a collection of bits there that are likely to be invalid as an instruction, causing the computer to crash. So we need to tell it to `halt` after it's done with its work.

That's it! But will our program work

Trying It Out

It would be hmmmurous if we got this wrong

We can *assemble* the program by running `hmmmAssembler.py` from the command line: [1] User-typed input is shown in blue and the prompt is shown using the symbol `%`. The prompt on your computer may look different.

```
% ./hmmmAssembler.py
Enter input file name: triangle1.hmmm
Enter output file name: triangle1.hb

----------------------
| ASSEMBLY SUCCESSFUL |
----------------------

0 : 0000 0001 0000 0001        0        read    r1       # Get base
1 : 0000 0010 0000 0001        1        read    r2       # Get height
2 : 1000 0001 0001 0010        2        mul     r1 r1 r2 # b times h into
3 : 0001 0010 0000 0010        3        setn    r2 2
4 : 1001 0001 0001 0010        4        div     r1 r1 r2 # Divide by 2
5 : 0000 0001 0000 0010        5        write   r1
6 : 0000 0000 0000 0000        6        halt
```

If you have errors in the program, `hmmmAssembler.py` will tell you; otherwise it will produce the output file `triangle1.hb` ("hb" stands for "Hmmm binary"). We can then test our program by running the Hmmm simulator:

```
% ./hmmmSimulator.py
Enter binary input file name: triangle1.hb
Enter debugging mode? n
Enter number: 4
Enter number: 5
10
% ./hmmmSimulator.py
Enter binary input file name: triangle1.hb
Enter debugging mode? n
Enter number: 5
Enter number: 5
```

We can see that the program produced the correct answer for the first test case, but not for the second. That's because Hmmm only works with integers; division rounds fractional values down to the next smaller integer just as integer division does in Python.

4.5.2 Looping

If you want to calculate the areas of a lot of triangles, it's a nuisance to have to run the program over and over. Hmmm offers a solution in the *unconditional jump* instruction, which says "instead of executing the next sequential instruction, start reading instructions beginning at address *n* ." If we simply replace the `halt` with a jump:

```
#
# Calculate the approximate areas of many triangles.
#
# First input: base
# Second input: height
# Output: area
#

0       read    r1       # Get base
1       read    r2       # Get height
2       mul     r1 r1 r2 # b times h into r1
3       setn    r2 2
4       div     r1 r1 r2 # Divide by 2
5       write   r1
6       jumpn   0
```

then our program will calculate triangle areas forever.

What's that `jumpn 0` instruction doing? The short explanation is that it's telling the computer to jump back to location 0 and continue executing the program there. The better explanation is that this instruction simply puts the number 0 in the program counter. Remember, the computer mindlessly checks its program counter to determine where to fetch its next instruction from memory. By placing a 0 in the program counter, we are ensuring that the next time the computer goes to fetch an instruction it will fetch it from memory location 0.

Since there will come a time when we want to stop, we can *force* the program to end by holding down the Ctrl ("Control") key and typing C (this is commonly written "Ctrl-C" or just "^C"'):

```
% ./hmmSimulator.py
Enter binary input file name: triangle2.hb
Enter debugging mode? n
Enter number: 4
Enter number: 5
10
Enter number: 5
Enter number: 5
12
Enter number: ^C

Interrupted by user, halting program execution...
```

That works, but it produces a somewhat ugly message at the end. A nicer approach might be to automatically halt if the user inputs a zero for either the base or the height. We can do that with a *conditional jump* instruction, which works like `jumpn` if some *condition* is true, and otherwise does nothing.

There are several varieties of conditional jump statements and the one we'll use here is called `jeqzn` which is pronounced "jump to *n* if equal to zero" or just "jump if equal to zero." This conditional jump takes a register and a number as input. If the specified register contains the value zero then we will jump to the instruction specified by the number in the second argument. That is, if the register contains zero then we will place the number in the second argument in the program counter so that the computer will continue computing using that number as its next instruction.

I believe that I learned that when someone jeqzn's I should say "gesundheit"

```
#
# Calculate the approximate areas of many triangles.
# Stop when a base or height of zero is given.
#
# First input: base
# Second input: height
# Output: area
#

0       read    r1       # Get base
1       jeqzn   r1 9     # Jump to halt if base is zero
2       read    r2       # Get height
3       jeqzn   r2 9     # Jump to halt if height is zero
4       mul     r1 r1 r2 # b times h into r1
5       setn    r2 2
6       div     r1 r1 r2 # Divide by 2
7       write   r1
8       jumpn   0
```

```
9       halt
```

Now, our program behaves politely:

```
% ./hmmmSimulator.py
Enter binary input file name: triangle3.hb
Enter debugging mode? n
Enter number: 4
Enter number: 5
10
Enter number: 5
Enter number: 5
12
Enter number: 0
```

The nice thing about conditional jumps is that you aren't limited to just terminating loops; you can also use them to make decisions. For example, you should now be able to write a Hmmm program that prints the absolute value of a number. The other conditional jump statements in Hmmm are included in the listing of all Hmmm instructions at the end of this chapter.

4.5.3 Functions

Here is a program that computes factorials:

```
#
# Calculate N factorial.
#
# Input: N
# Output: N!
#
# Register usage:
#
#       r1      N
#       r2      Running product
#

0       read    r1       # Get N
1       setn    r2,1
2       jeqzn   r1,6     # Quit if N has reached zero
3       mul     r2,r1,r2 # Update product
4       addn    r1,-1    # Decrement N
5       jumpn   2        # Back for more

6       write   r2
```

```
7       halt
```

(If you give this program a negative number, it will crash unpleasantly; how could you fix that problem?) The `addn` instruction at line 4 simply adds a constant to a register, replacing its contents with the result. We could achieve the same effect by using `setn` and a normal `add`, but computer programs add constants so frequently that Hmmm provides a special instruction to make the job easier.

But suppose you need to write a program that computes $\binom{n}{k} = \frac{n!}{k!(n-k)!}$. (If you haven't seen this formula before, it counts the number of different ways to choose k items from a set of n distinct items and it's pronounced "n choose k".)

Since we need to compute three different factorials, we would like to avoid having to copy the above loop three different times. Instead, we'd prefer to have a *function* that computes factorials, just like Python has.

I wouldn't choose to write that program.

Creating a function is just a bit tricky. It can't read its input from the user, and it must return the value that it computed to whatever code called that function.

It's convenient to adopt a few simple conventions to make this all work smoothly. One such convention is to decide on special registers to be used for *parameter passing*, i.e., getting information into and out of a function. For example, we could decide that r1 will contain n when the factorial function starts, and r2 will contain the result. (As we'll see later, this approach is problematic in the general case, but it's adequate for now.)

Our new program, with the factorial function built in, is:

```
#
# Calculate C(n,k) = n!/k!(n-k)!.
#
# First input: N
# Second input: K
# Output: C(N,K)
#
# Register usage:
#
#       r1      Input to factorial function
#       r2      r1 factorial
#       r3      N
#       r4      K
#       r5      C(N,K)
```

```
#
# Factorial function starts at address 15
#

0       read    r3       # Get N
1       read    r4       # Get K

2       copy    r1,r3    # Calculate N!
3       calln   r14,15   # ...
4       copy    r5,r2    # Save N! as C(N,K)

5       copy    r1,r4    # Calculate K!
6       calln   r14,15   # ...
7       div     r5,r5,r2 # N!/K!

8       sub     r1,r3,r4 # Calculate (N-K)!
9       calln   r14,15   # ...
10      div     r5,r5,r2 # C(N,K)

11      write   r5       # Write answer
12      halt

13      nop              # Waste some space
14      nop

# Factorial function.  N is in R1. Result is R2.
# Return address is in R14.
15      setn    r2,1     # Initial product
16      jeqzn   r1,20    # Quit if N has reached zero
17      mul     r2,r1,r2 # Update product
18      addn    r1,-1    # Decrement N
19      jumpn   16       # Back for more

20      jumpr   r14      # Done; return to caller
```

As you can see, the program introduces a number of new instructions. The simplest is `nop`, the *no-operation* instruction, at lines 13 and 14. When executed, it does absolutely nothing.

Why would we want such an instruction? If you've already written some small Hmmm programs, you've probably discovered the inconvenience of renumbering lines. By including a few `nop`s as *padding*, we make it easy to insert a new instruction in the sequence from 0-15 without having to change where the factorial function begins.

Far more interesting is the `calln` instruction, which appears at lines 3, 6, and 9. `Calln` works similarly to `jumpn`: it causes Hmmm to start executing instructions at a given address, in this case 15. But if we had just used a `jumpn`, after the factorial function

One of the hmmmdingers of programming in

calculated its result, it wouldn't know whether to jump back to line 4, line 7, or line 10! To solve the problem, the `calln` uses register R14 to save the address of the instruction immediately *following* the call. [2]

assembly language!

We're not done, though: the factorial function itself faces the other end of the same dilemma. R14 contains 4, 7, or 10, but it would be clumsy to write code equivalent to "If R14 is 4, jump to 4; otherwise if R14 is 7, jump to 7; ..." Instead, the `jumpr` (jump to address in register) instruction solves the problem neatly, if somewhat confusingly. Rather than jumping to a fixed address given in the instruction (as is done in line 19), `jumpr` goes to a *variable* address taken from a register. In other words, if R14 is 4, `jumpr` will jump to address 4; if it's 7 it will jump to 7, and so forth.

4.5.4 Recursion

In Chapter 2 we learned to appreciate the power and elegance of recursion. But how can you do recursion in Hmmm assembly language? There must be a way; after all, Python implements recursion as a series of machine instructions on your computer. To understand the secret, we need to talk about the *stack*.

Stacks

You may recall that in Chapter 2 we talked about stacks (remember those stacks of storage boxes)? Now we're going to see precisely how they work.

A stack is something we are all familiar with in the physical world: it's just a pile where you can only get at the top thing. Make a tall stack of books; you can only see the cover of the top one, you can't remove books from the middle (at least not without risking a collapse!), and you can't add books anywhere except at the top.

The sheer weight of that book could crush your stack!

The reason a stack is useful is because it remembers things and gives them back to you in reverse order. Suppose you're reading *Gone With the Wind* and you start wondering whether it presents the American Civil War fairly. You might put *GWTW* on a stack, pick up a history book, and start researching. As you're reading the history book, you run into an unfamiliar word. You place the history book on the stack and pick up a dictionary. When you're done with the dictionary, you return it to the shelf, and the top of the stack has the history book, right where you left off. After you finish your research, you

put that book aside, and voilá! *GWTW* is waiting for you.

This technique of putting things aside and then returning to them is precisely what we need for recursion. To calculate $5!$, you need to find $4!$ for which you need $3!$ and so forth. A stack can remember where we were in the calculation of $4!$ and help us to return to it later.

To implement a stack on a computer, we need a *stack pointer*, which is a register that holds the memory address of the top item on the stack. Traditionally, the stack pointer is kept in the highest-numbered register, so on Hmmm we use R15.

Again, this is just a convention.

Suppose R15 currently contains 102, [3] and the stack contains 42 (on top), 56, and 12. Then we would draw the stack like this:

	Address	Contents
R15 →	102	42
	101	56
	100	12

To *push* a value, say 23, onto the stack, we must increment R15 to contain the address of a new location (103) and then store 23 into that spot in memory.

Later, to *pop* the top value off, we must ask R15 where the top is [4] (103) and recover the value in that location. Then we decrement R15 so that it points to the new stack top, location 102. Everything is now back where we started.

The code to push something, say the contents of R4, on the stack looks like this:

```
addn r15,1      # Point to a new location
storer r4,r15   # Store r4 on the stack
```

The `storer` instruction stores the contents of R4 into the memory location *addressed* by register 15. In other words, if R15 contains 103, the value in R4 will be copied into memory location 103.

Just as `storer` can put things onto the stack, `loadr` will recover them:

```
loadr r3,r15    # Load r3 from the stack
```

```
addn r15,-1     # Point to new top of stack
```

Important note: in the first example above, the stack pointer is incremented *before* the `storer`; in the second, it is decremented *after* the *loadr*`. This ordering is necessary to make the stack work properly; you should be sure you understand it before proceeding further.

Saving Precious Possessions

When we wrote the $\binom{n}{k}$ program in Section 4.5.3, we took advantage of our knowledge of how the factorial function on line 15 worked. In particular, we knew that it only modified registers R1, R2, and R14, so that we could use R3 and R4 for our own purposes. In more complex programs, however, it may not be possible to partition the registers so neatly. This is especially true in recursive functions, which by their nature tend to re-use the same registers that their callers use.

The only way to be sure that a called function won't clobber your data is to save it somewhere, and the stack is a perfect place to use for that purpose. When writing a Hmmm program, the convention is that you must save all of your "precious possessions" before calling a function, and restore them afterwards. Because of how a stack works, you have to restore things in reverse order.

But what's a precious possession? The quick answer is that it's any register that you plan to use, *except* R0 and R15. In particular, if you are calling a function from inside another function, R14 is a precious possession.

It's a common mistake, but to err is hmmman.

Many newcomers to Hmmm try to take shortcuts with stack saving and restoring. That is to reason, "I know that I'm calling two functions in a row, so it's silly to restore my precious possessions and save them again right away." Although you can get away with that trick in certain situations, it's very difficult to get it right, and you are much more likely to cause yourself trouble by trying to save time. We strongly suggest that you follow the suggested *stack discipline* rigorously to avoid problems.

Let's look at a Hmmm program that uses the stack. We'll use the recursive algorithm to calculate factorials:

```
# Calculate N factorial, recursively
```

```
#
# Input: N
# Output: N!
#
# Register usage:
#
#       r1      N! (returned by called function)
#       r2      N

0       setn    r15,100  # Initialize stack pointer
1       read    r2       # Get N
2       calln   r14,5    # Recursive function finds N!
3       write   r1       # Write result
4       halt

# Function to compute N factorial recursively
#
# Inputs:
#
#       r2      N
#
# Outputs:
#
#       r1      N!
#
# Register usage:
#
#       r1      N! (from recursive call)
#       r2      N (for multiplication)

5       jeqzn   r2,18    # Test for base case (0!)

6       addn    r15,1    # Save precious possessions
7       storer  r2,r15   # ...
8       addn    r15,1    # ...
9       storer  r14,r15  # ...

10      addn    r2,-1    # Calculate N-1
11      calln   r14,5    # Call ourselves recursively to get (N-1)!

12      loadr   r14,r15  # Recover precious possessions
13      addn    r15,-1   # ...
14      loadr   r2,r15   # ...
15      addn    r15,-1   # ...

16      mul     r1,r1,r2 # (N-1)! times N
17      jumpr   r14      # Return to caller

# Base case: 0! is always 1
18      setn    r1,1
19      jumpr   r14      # Return to caller
```

The main program is quite simple (lines 0 - 4): we read a number from the user, call the recursive factorial function, and write the answer, and then halt.

The factorial function is only a bit more complex. After testing for the base case of $0!$, we save our "precious possessions" in preparation for the recursive call (lines 6 - 9). But what is precious to us? The function uses registers R1, R2, R14, and R15. We don't need to save R15 because it's the stack pointer, and R1 doesn't yet have anything valuable in it. So we only have to save R2 and R14. We follow the stack discipline, placing R2 on the stack (line 7) and then R14 (line 9).

After saving our precious possessions, we call ourselves recursively to calculate $(N-1)!$ (lines 10–11) and then recover registers R14 (line 12) and R2 (line 14) in reverse order, again following the stack discipline. Then we multiply $(N-1)!$ by N, and we're done.

It is worth spending a bit of time studying this example to be sure that you understand how it operates. Draw the stack on a piece of paper, and work out how values get pushed onto the stack and popped back off when the program calculates $3!$.

4.5.5 The Complete Hmmm Instruction Set

This finishes our discussion of Hmmm. We have covered all of the instructions except `sub`, `mod`, `jnezn`, `jgtzn`, and `jltzn`; we trust that those don't need separate explanations.

Note that `sub` *can be combined with* `jltzn` *to evaluate expressions like* $a < b$.

For convenience, Figure 4.10 at the bottom of the page summarizes the entire instruction set, and also gives the binary encoding of each instruction.

As a final note, you may find it instructive to compare the encodings of certain pairs of instructions. In particular, what is the difference between `add`, `mov`, and `nop`? How does `calln` relate to `jumpn`? We will leave you with these puzzles as we move on to imperative programming.

4.5.6 A Few Last Words

What actually happens when we run a program that we've written in a language such as Python? In some systems, the entire program is first automatically translated or *compiled* into machine language (the binary equivalent of assembly language) using another

program called a *compiler*. The resulting compiled program looks like the Hmmm code that we've written and is executed on the computer. Another approach is to use a program called an *interpreter* that translates the instructions one line at a time into something that can be executed by the computer's hardware.

It's important to keep in mind that exactly when and how the program gets translated - all at once as in the case of a compiler or one line at a time in the case of an interpreter - is an issue separate from the language itself. In fact, some languages have both compilers and interpreters, so that we can use one or the other.

Why would someone even care whether their program was compiled or interpreted? In the compiled approach, the entire program is converted into machine language before it is executed. The program runs very fast but if there is an error when the program runs, we probably won't get very much information from the computer other than seeing that the program "crashed". In the interpreted version, the interpreter serves as a sort of "middle-man" between our program and the computer. The interpreter can examine our instruction before translating it into machine language. Many bugs can be detected and reported by the interpreter before the instruction is ever actually executed by the computer. The "middle-man" slows down the execution of the program but provides an opportunity to detect potential problems. In any case, ultimately every program that we write is executed as code in machine language. This machine language code is decoded by digital circuits that execute the code on other circuits.

4.6 Conclusion

Aye caramba! That was a lot. We've climbed the levels of abstraction from transistors, to logic gates, to a ripple-carry adder, and ultimately saw the general idea of a how a computer works. Finally, we've programmed that computer in its native language.

Now that we've seen the foundations of how a computer works, we're going back to programming and problem-solving. In the next couple of chapters, some of the programming concepts that we'll see will be directly tied to the issues that we examined in this chapter. We hope that the understanding that you have for the internal workings of a computer will help the concepts that we're about to see be even more meaningful than they would be otherwise.

I think a mochaccino and a donut would be quite meaningful right about now.

Instruction	Description	Aliases

System instructions		
halt	Stop!	None
read rX	Place user input in register rX	None
write rX	Print contents of register rX	None
nop	Do nothing	None
Setting register data		
setn rX N	Set register rX equal to the integer N (-128 to +127)	None
addn rX N	Add integer N (-128 to 127) to register rX	None
copy rX rY	Set rX = rY	**mov**
Arithmetic		
add rX rY rZ	Set rX = rY + rZ	None
sub rX rY rZ	Set rX = rY - rZ	None
neg rX rY	Set rX = -rY	None
mul rX rY rZ	Set rX = rY * rZ	None
div rX rY rZ	Set rX = rY / rZ (integer division; no remainder)	None
mod rX rY rZ	Set rX = rY % rZ (returns the remainder of integer division)	None
Jumps!		
jumpn N	Set program counter to address N	None
jumpr rX	Set program counter to address in rX	**jump**
jeqzn rX N	If rX == 0, then jump to line N	**jeqz**
jnezn rX N	If rX != 0, then jump to line N	**jnez**
jgtzn rX N	If rX > 0, then jump to line N	**jgtz**
jltzn rX N	If rX < 0, then jump to line N	**jltz**
calln rX N	Copy the next address into rX and then jump to mem. addr. N	**call**
Interacting with memory (RAM)		
loadn rX N	Load register rX with the contents of memory address N	None
storen rX N	Store contents of register rX into memory address N	None
loadr rX rY	Load register rX with data from the address location held in reg. rY	**loadi, load**
storer rX rY	Store contents of register rX into memory address held in reg. rY	**storei, store**

Figure 4.10: The Hmmm instruction set.

Footnotes

[1] On Windows, you can navigate to the command line from the Start menu. On a Macintosh, bring up the Terminal application, which lives in Applications/Utilities.

On Linux, most GUIs offer a similar terminal application in their main menu. The example above was run on Linux.

[2] We could have chosen any register except R0 for this purpose, but by convention Hmmm programs use R14.

[3] We say that R15 *points to* location 102.

[4] We say we *follow the pointer* in R15.

Chapter 5: Imperative Programming

> *The imperative is to define what is right and do it.*
>
> —Barbara Jordan

5.1 A Computer that Knows You (Better than You Know Yourself?)

These days it seems that just about every website out there is trying to recommend something. Websites recommend books, movies, music, and activities. Some even recommend friends!

How does Netflix know what movies we're likely to enjoy watching or Amazon know what we'll like to read?

The answer to this question lies in a broad and important subfield of computer science: *data mining*. Data mining focuses on extracting useful information from very large quantities of unstructured data.

Netflix recommended the movie "Alien" to me.

In this chapter we will examine a simplified version of a fundamental data mining algorithm called *collaborative filtering (CF)*. Collaborative filtering is widely used in many successful recommendation systems, as well as many other application areas where there's a lot of data to be studied (e.g., financial markets, geological data, biological data, web pages, etc.). Our goal in this chapter is to build a music recommender system that uses a basic version of the CF approach.

Of course, our recommender system and our CF approach will necessarily be simplified. To build an industrial-strength recommender system is quite complicated. The system that won the Netflix prize (see http://www.netflixprize.com for more details) consists of several different sophisticated algorithms. See the paper "The BellKor Solution to the Netflix Grand Prize" by Yehuda Koren at http://www.netflixprize.com/assets/GrandPrize2009_BPC_BellKor.pdf

To motivate the idea behind collaborative filtering, let's go back to the dark ages before the World Wide Web became pervasive. In those ancient times – perhaps before you

were born – people discovered movies, books, restaurants, and music by asking for recommendations from people whom they knew to have tastes similar to their own.

Collaborative filtering systems work on this same fundamental principle–they find people who generally like the same things you do, and then recommend the things that they like that you might not have discovered yet. Actually, this is only one type of collaborative filtering called *user-based CF*. If you're dying to learn more about different types of CF, good! Keep taking CS courses. Or for a more immediate gratification, read the Wikipedia page: http://en.wikipedia.org/wiki/Collaborative_filtering

Let's consider a simple example in which a system is trying to recommend music you might like. First, the system needs to know something about your tastes, so you have to give it some information about what you like. Let's say you tell the system you like Lady Gaga, Katy Perry, Justin Bieber and Maroon 5.

The system knows about the following additional five "stored" users (users for which we've already stored some data):

I talked to your roommate and learned that those are your favorite groups, so don't try to deny it!

- April likes Maroon 5, The Rolling Stones and The Beatles
- Ben likes Lady Gaga, Adele, Kelly Clarkson, The Dixie Chicks and Lady Antebellum
- Cory likes Kelly Clarkson, Lady Gaga, Katy Perry, Justin Bieber and Lady Antebellum
- Dave likes The Beatles, Maroon 5, Eli Young Band and Scotty McCreery
- Edgar likes Adele, Maroon 5, Katy Perry, and Bruno Mars.

Determining a recommendation for you involves two steps:

- Determine which user (or users) have tastes most similar to yours.
- Choose and rank artists that the similar user likes. These are your recommendations.

Of course, there are many algorithms for performing each of the above steps, and many interesting problems that can arise in performing them, but again we will keep things relatively simple here. Our system will use the following simple CF algorithm:

- For each user, count the number of matching artists between your profile and the profile of that user.
- Choose the user with the highest number of matches.
- Recommend the artists who appear on their list and not on yours.

So in the above example, the system determines that Cory has the greatest overlap with you: He likes three of the same artists you like (Lady Gaga, Katy Perry and Justin Bieber) whereas none of the others have more than two groups in common with you. So, the recommendation system chooses Cory as the person whose tastes are most similar to yours. It also sees that Cory likes Lady Antebellum and Kelly Clarkson and therefore recommends that you check out these additional artists.

This is an admittedly simplified example and you can probably come up with a number of good questions about this approach. For example:

- What if I already know that I don't like Kelly Clarkson? How could the system use this information?
- What if there are multiple people who tie for the best match with my tastes? Conversely, what if no one matches my tastes?
- In a real system, there are millions of users. How can the system efficiently find the best match?

Most of these answers are beyond the scope of this chapter, but we encourage you to think about how *you* might answer these questions as you read through the rest of this chapter. If you're intrigued, great! This is exactly the kind of stuff you'll learn as you continue your CS studies. But for now, back to the foundations for our music recommendation system.

5.1.1 Our Goal: A Music Recommender System

In the rest of this chapter we will build a music recommender system that implements the basic collaborative filtering algorithm described above. The transcript below shows an example of an interaction with the system.

```
Welcome to the music recommender system!
What is your name? Christine

Welcome Christine.  Since you are a new user, I will need to gather some

Please enter an artist or band that you like: Maroon 5

Please enter another artist or band that you like,
or just press enter to see your recommendations: Lady Gaga

Please enter another artist or band that you like,
or just press enter to see your recommendations: Katy Perry
```

```
Please enter another artist or band that you like,
or just press enter to see your recommendations: Justin Bieber

Please enter another artist or band that you like,
or just press enter to see your recommendations:

Christine, based on the users I currently know about,
I believe you might like:
Lady Antebellum
Kelly Clarkson

I hope you enjoy them!  I will save your preferred artists and will have
```

The system keeps track of users and their preferences. As more users rate more songs, the system can generate additional recommendations, both for new and existing users. So, the next time Christine uses the system, she's likely to get some new recommendations.

Before we introduce the techniques that we'll use to implement the recommendation system, take a few moments to think about the steps involved in the program demonstrated above. What data do you need to gather and store? How do you need to process the data? What do you need to produce as output?

Although there are many ways to actually write the program, the underlying components are the following:

- The ability to gather input from the user;
- The ability to repeat a task (e.g., asking for the user's preferred artists, comparing against stored users' preferences) many times;
- The ability to store and manipulate data in different ways (e.g., storing the user's responses, manipulating the stored users' preferences); and
- The ability to save data between program runs, and to load saved data (e.g., loading the stored users' preferences from a file, saving the current user's preferences to a file).

Throughout the rest of this chapter we will learn ways to perform each of the above tasks, ultimately resulting in a fully-functional music recommender system.

5.2 Getting Input from the User

The first task that we have listed above is to get input from the user. Fortunately, this is a

simple task thanks to Python's built-in `raw_input` function. (This function is called `input` in Python 3.) Here's an example of some code that uses the `raw_input` function:

```
def greeting():
    name = raw_input('What is your name? ')
    print ('Nice to meet you,', name)
```

And here's how that code looks when it is run:

```
>>> greeting()
What is your name? Christine
Nice to meet you, Christine
```

The function `raw_input` takes one argument: a string that will be printed when the `raw_input` function is run (e.g., the string `'What is your name?'` in the example above). When Python sees the `raw_input` command, it prints that string and then waits for the user to type in her own string. The user types in a string and presses Return (or Enter). At that point, Python assigns the user's input string to the variable appearing on the left-hand side of the = sign. In the example above, the string that the user typed in will be assigned to the variable `name`.

Python's `raw_input` function will always return a string. If you want that string to be converted into something else, like an integer or a floating point number (a number with a decimal point – also known as a "float"), that can be done using Python's data conversion functions. In the example below, we take the user's input string and convert it into a floating point number.

```
def plus42():
    number = float(raw_input('Enter a number: '))
    nPlus42 = number + 42
    print (number, '+ 42 =', nPlus42)

plus42()
```

Run

In the shell, this should appear as:

```
>>> plus42()
Enter a number: 15
15 + 42 = 57.0
```

Note that converting data from one type to another (e.g., from strings to floats) can be "risky." In the example above, we converted the user's input string "15" into the floating point number 15.0. However, had the user entered the string "watermelon", Python's attempt to convert that to a floating point number would have failed and resulted in an error message.

Proving that watermelons don't float!

5.3 Repeated Tasks–Loops

The next feature that our program needs is to perform some set of actions multiple times.

For example, the recommender system needs to ask the user to enter a number of artists that they like. For the moment, let's change the system slightly so that instead of allowing the user to enter as many artists as she likes, the system asks the user to enter exactly four artists. The algorithm that performs this data-collection looks something like:

- Repeat the following four times:
    - Display a prompt to the user;
    - Obtain the user's input; and
    - Record the user's answer.

Actually, 42 times is just the right number of times for me!

Imagine that we have a few lines of Python code that gather and store input from the user. Since we want to do this four times, we could just make four copies of that code, one for each time that we want to ask the user for an artist. For only four preferences, this wouldn't be so bad. But imagine that we wanted to ask the user for 10 preferences? Or 42? The code would quickly become long and cumbersome. Furthermore, if we decided that we wanted to make a small change to the way that we ask the user for input, we'd have to make that change in 4, 10, or 42 places in our program. This would be very annoying for the programmer! Moreover, ultimately we might want to let the user herself specify how many artists she wants to enter, instead of fixing that value before the program runs. In that case, we simply wouldn't know in advance how many copies of the code to place in our program.

We have already seen one way to perform repeated tasks in Chapter 3. In fact, we've also seen a second: list comprehensions. This chapter presents another way to perform these same repeated tasks: *iteration* or *loops*.

Just as recursion allows a programmer to express self-similarity in a natural way, loops naturally express sequential repetition. It turns out that recursion and loops are often equally good choices for doing something repeatedly, but in many cases it's much easier and more natural to use one rather than the other. More on that shortly!

5.3.1 Recursion vs. Iteration at the Low Level

If you skipped

To illustrate the difference between iteration and recursion, let's travel backwards in time and revisit the recursive factorial function:

```python
def factorial(n):
    if n == 0:
        return 1
    else:
        answer = factorial(n - 1)
        return n * answer
```

Chapter 4, no worries– you might jump to Section 5.3.2

In Chapter 4, we implemented this function in HMMM and saw that, at the machine level, the recursion uses a stack to keep track of the function calls and the values of the variables in each function call. In that chapter, we also looked at another very different implementation of the factorial function that was considerably simpler and did not use a stack. Here it is:

```
#
# Calculate N factorial
# Input: N
# Output: N!
#
# Register usage:
#
#       r1      N
#       r2      Running product
#

00 read r1          # read n from the user
01 loadn r13 1      # load 1 into the return register
                    # (base case return value)
02 jeqz r1 06       # if we are at the base case, jump to the
                    # end
03 mul r13 r13 r1   # else multiply the answer by n
04 addn r1 -1       # and decrement n
05 jump 02          # go back to line 2 and test for base case
                    # again
06 write r13        # we're done so print the answer
07 halt             # and halt
```

In contrast to the recursive version, this implementation uses a variable (`r13` in this example) to gradually "accumulate" the answer. Initially, `r13` is set to 1. In lines 3–5, we multiply `r13` by the current value of `n` (stored in register `r1`), decrement `n` by 1, and jumps back to line 2 to test if we should do this again. We repeat this loop until the value of `n` reaches 0. At that point, `r13` contains the product $n \times (n-1) \times \cdots \times 1$ and we're done.

In some sense, this iterative approach is simpler than recursion because it just loops "round and round", updating the value of a variable until it's done. There's no need to

keep track (on the stack) of what "things were like" before the recursive call and reinstate those "things" (from the stack) when the recursion returns. Let's now see how this works in Python.

5.3.2 Definite Iteration: `for` loops

In many cases, we want to repeat a certain computation a specific number of times. Python's `for` loop expresses this idea. Let's take a look at how we can use a `for` loop to implement the iteration in the recommendation program to collect a fixed number of artists that the user likes.

To begin, assume the user will enter three artists.

```
artists = []

for i in [0, 1, 2]:
    next_artist = raw_input('Enter an artist that you like:')
    artists.append(next_artist)

print ('Thank you!  We'll work on your recommendations now.')
```

In Python 3, remember to use `input` instead of `raw_input`.

This `for` loop will repeat its "body", lines 4–5, exactly 3 times. Let's look closely at how this works. Line 3 above is the *header* of the loop, with five required parts:

1. The keyword `for`;
2. The name of a variable that will control the loop. In our case that variable is named `i`. It's safest to use a new variable name, created just for this `for` loop, or one whose old value is no longer needed;
3. The keyword `in`;
4. A sequence such as a list or a string. In our case it is the list `[0, 1, 2]`; and
5. A colon. This is the end of the header and the start of the `for` loop body.

As we noted, lines 4–5 are called the *body* of the loop. The instructions in the loop body must be indented consistently within the loop header, just as statements within functions are. Note that line 7 above is *not* part of the loop body because it is not indented.

So what exactly is going on in this `for` loop? Our variable `i` will initially take the first value in the list, which is 0. It will then perform the lines of code that are indented below the `for`

loop. In our case, those are lines 4 and 5. Line 4 uses Python's `raw_input` function (or `input` in Python 3) which prints the string

```
Enter an artist that you like:
```

and then pauses to let the user type in a response. Once the user has typed a response and pressed the ENTER (or RETURN) key, that response is placed in the variable named `next_artist`. So, `next_artist` will be a string that is the name of an artist. Line 5 uses Python's `append` function to add that string to the list `artists`.

Python recognizes that this is the end of the body of the `for` loop because line 7 is not indented – that is, it's at the same level of indentation as line 3. So, Python now goes back to line 3 and this time sets `i` to be 1. Now, lines 4 and 5 are performed again, resulting in the user being asked for another string and that string being appended to the end of the `artists` list.

Again, Python goes back up to line 3 and this time `i` takes the value 2. Once again, Python executes lines 4 and 5 and we get another artist and add it to the `artists` list.

Finally, `i` has no more values to take since we told it to take values from the list `[0, 1, 2]`. So now the loop is done and Python continues computing at line 7.

Four fors!

How could we ask the user to respond to four examples instead of three? Easy! We could modify the loop header to be:

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

What about 25 iterations? Well, we could replace the list `[0, 1, 2, 3]` with the list `[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24]` but who wants to type all that? (We certainly didn't find it very much fun!) Instead, we can use the built-in `range` function (that we saw in Chapter 3) to automatically generate that list:

```
for i in range(25):
```

Remember that `range(25)` does indeed generate that 25-element list from 0 to 24.

Does the control variable have to be named `i`? No – any valid variable name can be used here. For clarity and to avoid confusion, you should generally avoid using a name that you have

already used earlier in your function, although we often reuse variable names in loops.

Finally, a word on style: In general, variable names like `i` are not very descriptive. However, a variable used in a loop is a very temporary variable – it serves the purpose of allowing us to loop and then once the loop is over we're done with it. Therefore, it's quite common to use short variable names in `for` loops, and the names `i`, `j`, and `k` are particularly popular.

5.3.3 How Is the Control Variable Used?

In the example above, the control variable took on a new value at every iteration through the loop, but we never explicitly *used* the value of variable inside the body of the loop. Therefore, it didn't matter what the list of elements in our for loop header was—these elements were ignored. We could have accomplished the same thing with any three-element list header:

```
for i in ['I', 'love', 'Spam']:
```

or

```
for i in [42, 42, 42]:
```

In both cases, we would have gone through the loop three times.

Sometimes the value of the control variable is important to the computation inside the `for` loop. For example, consider this iterative version of the factorial function. Here we've named the control variable `factor` to suggest its role:

```
def factorial(n):
    answerSoFar = 1
    for factor in range(1, n+1):
        answerSoFar = answerSoFar * factor
    return answerSoFar
```

When we call `factorial(4)`, the loop becomes:

```
for factor in range(1, 5):
```

```
    answerSoFar = answerSoFar * factor
```

What happens here?

- After the first time through the loop `answerSoFar` holds the result of its previous value (1) times the value of `factor` (1). When the loop completes its first iteration `answerSoFar` will be 1 (i.e., `1*1`).
- In the second iteration through the loop `answerSoFar` will again be assigned to hold the product of the previous value of `answerSoFar` times `factor`. Since `factor`'s value is now 2, `answerSoFar` will equal 1*2, or 2, when this second loop iteration ends.
- After the third time through the loop, `answerSoFar` will be 2*3 or 6, and the fourth time through `answerSoFar` will become :math: 4*6, or 24.

The loop repeats exactly 4 times because `range(1, 5)` has four elements: `[1, 2, 3, 4]`.

Let's now "unroll" this four-iteration loop to see what it does at each iteration. Calling `factorial(4)` becomes

```
# factorial function begins
answerSoFar = 1

# loop iteration 1
factor = 1
answerSoFar = answerSoFar * factor  # answerSoFar becomes 1
# iteration 2
factor = 2
answerSoFar = answerSoFar * factor  # answerSoFar becomes 2
# iteration 3
factor = 3
answerSoFar = answerSoFar * factor  # answerSoFar becomes 6
# iteration 4
factor = 4
answerSoFar = answerSoFar * factor  # answerSoFar becomes 24

# loop ends, 24 is returned
```

5.3.4 *Accumulating* Answers

Consider what happened to `answerSoFar` throughout the iterations above. It started with a value and it changed that value each time through the loop, so that when the loop

completed the returned `answerSoFar` was the final answer. This technique of *accumulation* is very common, and the variable that "accumulates" the desired result is called an *accumulator*.

Let's look at another example. The `listDoubler` function below returns a new list in which each element is double the value of its corresponding element in the input list, `aList`. Which variable is the accumulator here?

```
1  def listDoubler(aList):
2      doubledList = []
3      for elem in aList:
4          # append the value 2*elem to doubledLis
5          doubledList.append(2*elem)
6      return doubledList
7
8  print (listDoubler([20, 21, 22]))
```

<< First | < Back | Step 1 of 12 | Forward > | Last >>

line that has just executed

next line to execute

Frames Objects

(ch05_ld)

In this case, the accumulator is the list `doubledList` rather than a number. It's growing in length one element at a time instead of one factor at a time as in `factorial`.

Let's consider one more example where the loop control variable's value is important to the functionality of a loop. Returning to our recommender program, let's write a loop that calculates the number of matches between two lists. This function will be useful when we're comparing the current user's preferences to the stored preferences of other users to determine which stored user most closely matches the current user's preferences. For this problem, we have two lists of artist/band names, for example:

```
>>> userPrefs
['Maroon 5', 'Lady Gaga', 'Justin Bieber']
>>> storedUserPrefs
['Maroon 5', 'Kelly Clarkson', 'Lady Gaga', 'Bruno Mars']
```

Our first version of this function will implement the following algorithm:

- Initialize a counter to 0
- For each artist in the user's preferences:
- If that artist is in the stored user's preferences too, add one to count

This simple algorithm is implemented with the following Python code:

```
def numMatches(userPrefs, storedUserPrefs):
    ''' return the number of elements that match between
        the lists userPrefs and storedUserPrefs '''
    count = 0
    for item in userPrefs:
        if item in storedUsersPrefs:
            count += 1
    return count
```

In this function we've used a shorthand notation to increment the value of `count`.

The line

```
count += 1
```

Similar shorthands include `-=`, `*=`, and `/=`

is just shorthand for

```
count = count + 1
```

Notice that the code above is actually not specific to our music recommender function, but in fact can be used to compare any two lists and return the number of matching elements between them. For this reason, it's stylistically better to use generic list names for the parameters to this function – that is, we could replace the variable name `userPrefs` with a more general name such as `listA` and `storedUserPrefs` with `listB`.

```
def numMatches(listA, listB):
    ''' return the number of elements that match between
        listA and listB '''
    count = 0
    for item in listA:
        if item in listB:
            count += 1
```

```
    return count
```

Finally, let's complete the process of choosing the stored user with the highest number of matches to the current user. Let's assume that each stored user is represented by a list of that user's preferences and then we put all of those lists into one master list. Using the representation, the stored users' preferences might look like this:

```
[['Maroon 5', 'The Rolling Stones', 'The Beatles'],

    ['Lady Gaga', 'Adele', 'Kelly Clarkson', 'The Dixie Chicks',
    'Lady Antebellum'],

    ['Kelly Clarkson', 'Lady Gaga', 'Katy Perry', 'Justin Bieber',
    'Lady Antebellum'],

    ['The Beatles', 'Maroon 5', 'Eli Young Band', 'Scotty
    McCreery'],

    ['Adele', 'Maroon 5', 'Katy Perry', 'Bruno Mars']]
```

Note that we have not included the names of the stored users, since we only care about their preferences for now. However, we can think about the stored users as having indices. In the list above, index 0 corresponds to the user who likes `['Maroon 5', 'The Rolling Stones', 'The Beatles']`. There are four other users with indices 1, 2, 3, and 4. Let's will assume that this list does not contain the preferences of the current user.

So, imagine that we want to calculate the `index` of the stored user with the best match to our current user. For example, if our current user likes `['Lady Gaga', 'Katy Perry', 'Justin Bieber', 'Maroon 5']` then the stored user at index 2 in the stored user list above has the most matches in common – three matches, whereas all other stored users have fewer matches.

The algorithm for finding the user with the best match to the current user is the following:

- Initialize the maximum number of matches seen so far to 0
- For each stored user:
- Count the number of matches between that stored user's preferences and the current user's preferences
- If that number of matches is greater than the maximum number of matches so far:
- Update the maximum number of matches seen so far
- Keep track of index of that user

- Return the index of the user with the maximum number of matches

How could we express this in Python? A first attempt might begin like this:

```
def findBestUser(userPrefs, allUsersPrefs):
    ''' Given a list of user artist preferences and a
        list of lists representing all stored users'
        preferences, return the index of the stored
        user with the most matches to the current user. '''
    max_matches = 0 # no matches found yet!
    best_index = 0
    for pref_list in allUsersPrefs:
        curr_matches = numMatches(userPrefs, pref_list)
        if curr_matches > max_matches:
           # somehow get the index of pref_list??
```

Note that we maintain a variable called `max_matches` that stores the maximum number of matches found so far. Initially it is set to 0, because we haven't done any comparisons yet and have no matches. We also maintain a variable called `best_index` that stores the index of the list in the `allUsersPrefs` that has the highest number of matches. We've initialized this to 0 as well. Is this a good idea? It suggests that the list with index 0 has the best number of matches so far. In fact, we haven't even looked at the list with index 0 at this point, so this may seem a bit weird. On the other hand, it doesn't hurt to initialize this variable to 0 since we're about to begin counting matches in the loop and the best solution will be at least 0.

Notice that we reach a dead end in the last line of our function above. We have the element from the `allPrefs` list, but we have no way of getting the *index* that corresponds to it, and it's that index that we're committed to finding. Can you think of a way to solve this problem—before peeking at the solution below?

Here's our next attempt—this time, more successful:

```
def findBestUser(userPrefs, allUsersPrefs):
    ''' Given a list of user artist preferences and a
        list of lists represented all stored users'
        preferences, return the index of the stored
        user with the most matches to the current user. '''
    max_matches = 0
    best_index = 0
    for i in range(len(allUsersPrefs)):
        curr_matches = countMatches(userPrefs,
                                    allPrefs[i])
        if curr_matches > max_matches:
```

```
            best_index = i
            max_matches = curr_matches
    return best_index
```

What's the difference? Notice that now our `for` loop is iterating not over the lists in `allUsersPrefs` but rather over the indices of those lists. We do this using the `range` function to generate the list of all indices in `allUsersPrefs`. With this index variable, we can access the element in the list and we can store that index in our `best_index` variable.

5.3.5 Indefinite Iteration: `while` Loops

In all of the examples above, we knew exactly how many times we wanted to loop. However, in many cases, we *can't* know how many iterations we need — the number of iterations of the loop depends on some external factor out of our control.

Let's continue with our recommendation program. In the last section we collected a fixed number of preferred artists from the user. But in our original program we allowed the user to enter as many artists as they wished, whether that number was one or one thousand.

Recall that `for` loops always run a definite number of times, so this situation calls for a different kind of loop: the *while*. A `while` loop runs as long as its boolean condition is true.

Let's take a close look at the structure of a `while` loop that implements the new desired behavior for our recommender program

In other words, a `while` *loop runs for a* `while`.

```
newPref = raw_input("Please enter the name of an \
artist or band that you like: " )

while newPref != '':
    prefs.append(newPref)
    newPref = raw_input("Please enter an artist or ba
that you like, or just press enter to see recommendat

print('Thanks for your input!')
```

In Python 3, remember to use `input` rather than `raw_input`.

The `while` loop is similar to the `for` loop in that it consists of a loop header (line 4) and a loop body (lines 5-7). The loop header consists of the following three elements, in order:

- The keyword `while`
- A boolean expression. In our example this expression is `newPref != ''`
- A colon

As with `for` loops, the loop body must be indented under the loop header. Thus, line 9 in the above example is *not* inside the loop body. Note that both of the `raw_input` statements are split over more than one line of text, but Python considers each really just one line because of the textbackslash symbols at the end of the lines.

A `while` loop will execute as long as the Boolean expression in the header evaluates to `True`. In this case, the Boolean expression is `newPref != ''`. Assuming that the user entered a non-empty string in line 1, the expression `newPref != ''` will evaluate to `True` the first time it is evaluated. Thus, we enter the body of the `for` loop and execute lines 5–7. If the user entered another non-empty string, then this Boolean will again evaluate to `True`. This will repeat until, eventually, the user enters no string – she just presses RETURN or ENTER. At this point, `newPref` is an empty string. Thus, when we return to evaluate the Boolean expression `newPref != ''` in the `while` statement, it is now `False` and the loop terminates, causing execution to continue at line 9.

5.3.6 `for` Loops vs. `while` Loops

Often it's clear whether a computational problem calls for definite (`for`) or indefinite (`while`) iteration. One simple rule-of-thumb is that `for` loops are ideally suited to cases when you know exactly how many iterations of the loop will be conducted whereas `while` loops are ideal for cases when it's less clear in advance how many times the loop must repeat.

But aliens don't have thumbs!

It is always possible to use a `while` loop to emulate the behavior of a `for` loop. For example, we could express the factorial function with a `while` loop as follows:

```
def factorial(n):
    answer = 1
    while n > 0:
        answer = answer * n
        n = n-1
    return answer
```

Here we've used `answer` instead of `answerSoFar` with the

understanding that `answer` will not actually hold the value of the desired answer until the end of the loop–this is a common style for naming accumulator variables.

...that saves typing!

Be careful! Sometimes when people try to use a `while` loop to perform a specific number of iterations their code ends up looking like this:

```
def factorial(n):
    answer = 1
    while n > 0:
        answer = answer * n
    return answer
```

What happens when you run `factorial(5)` using the function above? The loop will run, but it will never stop!

This `while` loop depends on the fact that `n` will eventually reach 0, but in the body of the loop we never change the value of `n` and we never get `n!` Python will happily continue multiplying `answer * n` forever–or at least until you get tired of waiting and hit Ctrl-C to stop the program.

Go through the steps and see if you can spot the error in the following version:

```
1  def factorial(n):
2      answer = 1
3      while n > 0:
4          answer = answer * n
5      n = n-1
6      return answer
7
8  print (factorial(5))
```

<< First | < Back | Step 1 of 498 | Forward > | Last >>

(stopped after 500 steps to prevent possible infinite loop)

line that has just executed
next line to execute

Frames Objects

(ch05_fac)

This code will also run forever. But why? After all, we definitely do decrease `n`. This bug is more subtle. Remember that the `while` loop runs only the code in the body of the loop before repeating. Because the statement that subtracts 1 from `n` is not indented, it is not part of the `while` loop's body. Again, the loop variable does not change within the loop, and it runs forever.

A loop that never ends is called an *infinite loop* and it is a common programming bug. When using a `while` loop, remember to update your loop-control variable inside the loop. It's an advantage of `for` loops that this updating is done automatically! In fact, the tendency to accidentally create infinite loops is so common that it leads us to an important takeaway message about the choice between `for` and `while` loops.

The Apple Corporation's address is One Infinite Loop, Cupertino, CA

Takeaway message: *If you know in advance how many times you want to run a loop, use a* `for` *loop; if you don't, use a* `while` *loop.*

5.3.7 Creating Infinite Loops On Purposes

Sometimes infinite loops can come in handy. They're not actually infinite, but the idea is that we will stop them when we are "done," and we don't have to decide what "done" means until later in the loop itself.

For example, consider a different version of the recommender loop above that gathers data from the user.

I'm pro-crastination!

```
numCorrect = 0

while True:       # run forever -- or at least as long
newPref = raw_input("Please enter an artist or band th
                    or just press enter to see recommenda
if newPref:
    prefs.append(newPref)
else:
    break

print('Thanks for your input!')
```

The body of the `else` statement contains one instruction: the `break` instruction. `break` will

immediately halt the execution of the loop that it appears in, causing the code to jump to the next line immediately after the loop body. `break` can be used in any kind of loop, and its effect is always the same–if the code reaches a `break` statement, Python *immediately* exits the containing loop and proceeds with the next line after the loop. If you have one loop inside another loop (a perfectly OK thing to do, as you'll see below), the `break` statement exits only the innermost loop.

You might ask, "Do we really need a `break`?"

Yes! I need a `break`!

After all, the loop above can be written with a more informative condition, as we saw above. Which approach is better? It's a matter of style. Some prefer the "delayed decision" approach, writing loops that appear to run too long, only to break out of them from the inside; others prefer to put all of the conditions directly in the loop header. The advantage of the latter approach is that the condition helps clarify the context for the loop. The advantage of the former (in some cases anyway) is that it avoids the awkward double-input statement—there's no need to ask the user to enter their initial response in a separate input statement before the loop begins.

5.3.8 Iteration Is Efficient

The heart of imperative programming is the ability to change the value of variables—one or more accumulators—until a desired result is reached. These in-place changes can be more efficient, because they save the overhead of recursive function calls. For example, on our aging computer, the Python code

```
counter = 0
while counter < 10000:
    counter = counter + 1

ran in 2.6 milliseconds. The "equivalent" recursive program

def increment(value, times):
    if times <= 0:
        return value
    return increment(value + 1, times - 1)

counter = increment(0, 10000)
```

ran more than an order of magnitude slower, in 38.3 milliseconds.

Why the difference? Both versions evaluate 10,000 boolean tests; both execute the same 10,000 additions. The difference comes from the overhead of building and removing the stack frames used to implement the function calls made recursively.

Memory differences are even more dramatic: storing partial results on the stack can quickly exhaust even today's huge memory stores.

5.4 References and Mutable vs. Immutable Data

At the beginning of this chapter we introduced the two key components of imperative programming: iteration and data mutation. We have already discussed iteration in some depth. Now we begin to explore the concept of data mutation.

5.4.1 Assignment by *Reference*

The assignment and re-assignment of values to one variable—the accumulator—characterizes imperative programming with loops. All of these assignments are efficient, but **only if the size of the copied data is small!** Floating-point values and typical integers are stored in a small space, often the size of one register (32 or 64 bits). They can be copied from place to place rapidly. For example, the assignment operation

```
# suppose x refers to the value 42 right now
y = x
```

runs extremely fast, probably best measured in nanoseconds, as suggested by the timings in the previous section.

Lists, on the other hand, can grow very large. Consider this code:

```
# suppose that list1 holds the value of
# range(1000042) right now
list2 = list1
```

This assignment makes `list2` refer to a 1,000,042-element list—a potentially expensive proposition, if it involved 1,000,042 individual integer assignments like the `y = x` example above. What's more, there's no guarantee the elements of `list1` aren't lists themselvesldots .

How does Python make *both* integer and list assignments efficient? It does so through a simple, single rule for all of its data types: *Assignments copy a single reference.*

Reference? It turns out that when you assign a piece of data (e.g., an integer) to a variable, what you are actually doing is storing the memory location of that piece of data in the variable. We say that the variable holds a *reference* to the piece of data. When we think of a variable, we typically think only of the data to which the variable refers but you can also obtain the reference (i.e., the memory location of that data) with Python's built-in `id()` function:

```
>>> x = 42
>>> x            # Python will reply with x's value
42
>>> id(x)        # asks for x's reference (the memory location of its cont
505494448         # this will be different on your machine!
```

When we talk about a variable's *value*, we mean the data to which the variable refers; when we talk about a variable's *reference*, we mean the memory location of that data. For example, in the code above, the value of `x` is 42, and its reference is 505494448, which is the location of the value 42 in memory. By the way, the idea of the memory location of a piece of data should be very familiar to you from Chapter 4. If you like, you can think of the variables that store the references as registers on the CPU. (This is not quite exactly correct, but a reasonable conceptual model.) Figure 5.1 illustrates this concept graphically.

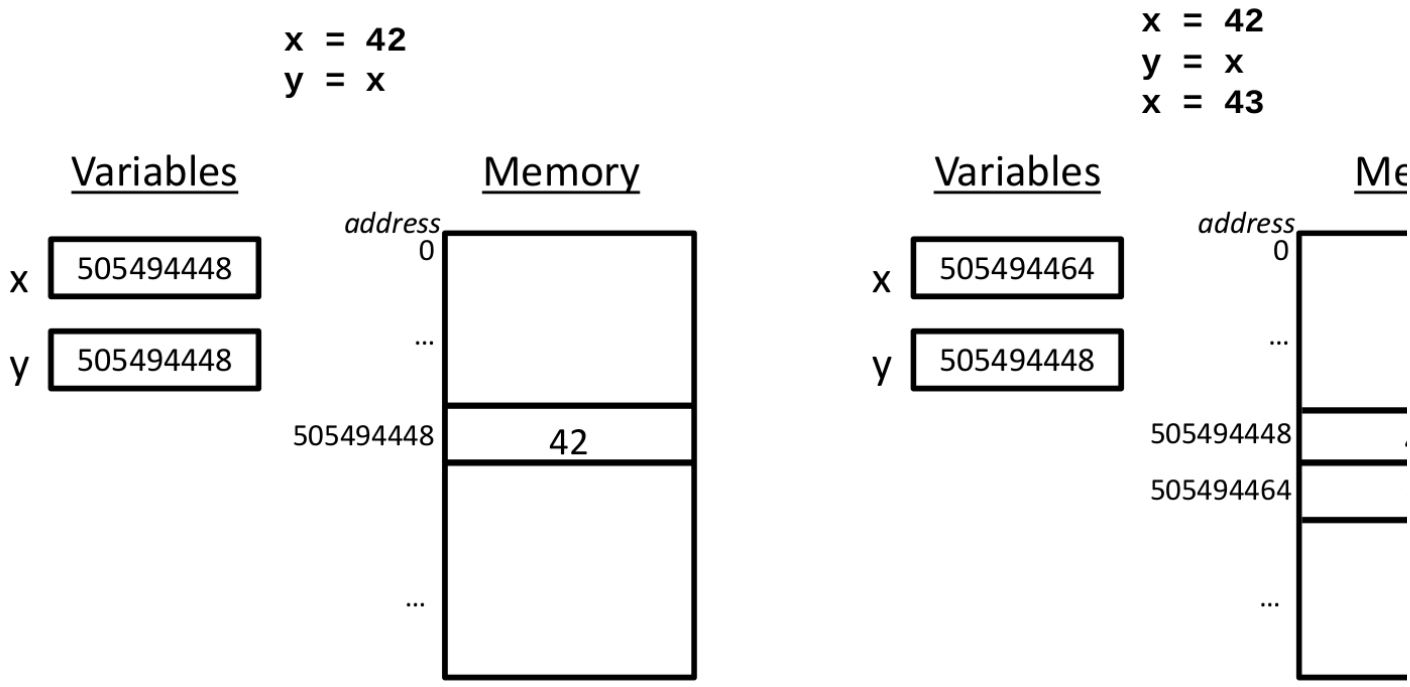


Figure 5.1: A depiction of how Python stores data. The boxes on the left are the variables, which you can think of as registers on the CPU, which store references to locations in memory. The actual data is stored in memory.

Python makes assignment efficient by *only copying the reference, and not the data*:

```
>>> x = 42      # this puts the value 42 in the next memory slot, 5054944
                # of that memory reference
>>> y = x       # copies the reference in x into y, so that x and y both
```

```
>>> id(x)       # asks for x's reference (the memory location of its cont
505494448
>>> id(y)       # asks for y's reference (the memory location of its cont
505494448
```

As you would expect, changes to `x` do not affect `y`:

```
>>> x = 43      # this puts 43 in the next memory slot,505494464
>>> id(x)
505494464        # x's reference has changed
>>> id(y)
505494448        # but y's has not
```

The result of executing the above code is shown on the right in Figure 5.1.

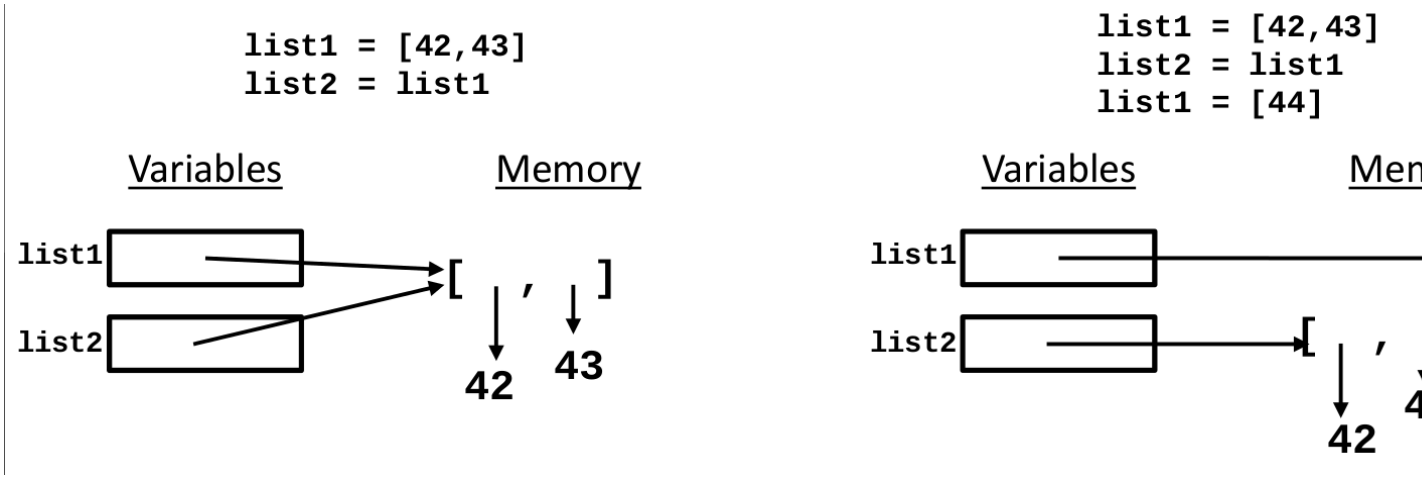


Figure 5.2: A box-and-arrow diagram abstraction of how Python stores list data. All of the elements in the list (and the list itself) are in memory, but we have abstracted away from the exact addresses of these elements.

Assignment happens the same way, regardless of the data types involved:

It is possible to reprogram how assignment works for user-defined data types, but this is the default.

```
>>> list1 = [42,43]  # this will create the list [42,43] and give its loc
>>> list2 = list1    # give list2 that reference, as well
```

```
>>> list1            # the values of each are as expected
[42,43]
>>> list2
[42,43]
```

```
>>> id(list1)        # asks for list1's reference (its memory location)
538664
>>> id(list2)        # asks for list2's reference (its memory location)
538664
```

As the data we store gets more complicated, it becomes cumbersome to try to actually represent what is happening in memory faithfully, so computer scientists often use a type of diagram called a box-and-arrow diagram to illustrate how references refer to memory. A box-and-arrow diagram for the above code is shown on the left in Figure 5.2. In this figure, the list `[42, 43]` exists in memory at the location referred to by both `list1` and `list2`, but instead of writing out the actual memory location, we indicate that `list1` and `list2` refer to the same data in memory using arrows. In fact, notice something interesting about Figure5.2: the elements in the list are also references to data in other locations in memory! This is a fundamental difference between integers and lists: a list refers to the memory location of a *collection* of potentially many elements, each of which has its own reference to the underlying data!

As with integers, if an assignment is made involving a list, *one reference is copied*, as shown in the code below, and on the right in Figure 5.2:

```
>>> list1 = [44]     # will create the list [44] and make list1 refer to
>>> id(list1)        # list1's reference has changed
541600
>>> id(list2)        # but list2's has not
```

```
538664
```

The assignment `list2 = list1` caused both `list1` and `list2` to refer to the same list.

This one-reference rule can have surprising repercussions. Consider this example, in which `x`, `list1[0]`, and `list2[0]` start out by holding the same reference to the value 42:

```
>>> x = 42           # to get started
>>> list1 = [x]      # similar to before
>>> list2 = list1    # give list2 that reference, as well
>>> id(x)        # all refer to the same data
505494448
>>> id(list1[0])
505494448
>>> id(list2[0])
505494448
```

What happens when we change `list1[0]`?

```
>>> list1[0] = 43    # this will change the reference held by the "zeroth
>>> list1[0]
43               # not surprising at all
>>> list2[0]
43               # aha! list1 and list2 refer to the same list
>>> x
42               # x is a distinct reference
```

Let's see

```
>>> id(list1[0])     # indeed, the reference of that zeroth element has c
505494464
>>> id(list2[0])     # list2's zeroth element has changed too!
505494464
>>> id(x)            # x is still, happily, the same as before
505494448
```

This example is illustrated in Figure 5.3 and codelens examples 'ch05_ref1' and 'ch05_ref2'.

```
1  x = 42
2  list1 = [x]
3  list2 = list1
```

<< First | < Back | Step 1 of 3 | Forward > | Last >>

line that has just executed
next line to execute

Frames Objects

(ch05_ref1)

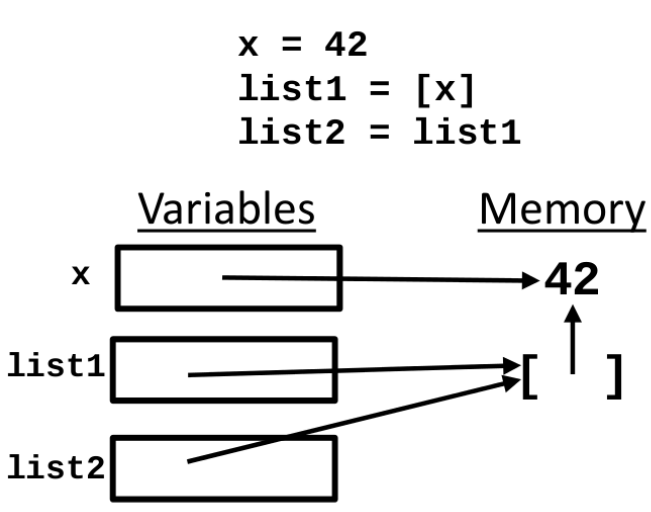
```
1  x = 42
2  list1 = [x]
3  list2 = list1
4  list1[0] = 43
```

<< First | < Back | Step 1 of 4 | Forward > | Last >>

line that has just executed
next line to execute

Frames Objects

(ch05_ref2)



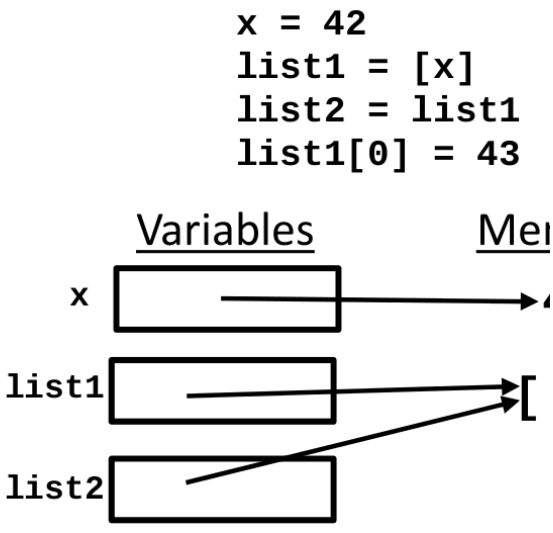


Figure 5.3: A graphical illustration of what happens when you modify the elements in a list that is referenced by two variables.

5.4.2 Mutable Data Types Can Be Changed Using Other Names!

Note that `list2[0]` has been changed in the above example, even though no assignment statements involving `list2[0]` were run! Because `list2` is really just another name for the exact same data as `list1`, we say that `list2` is an *alias* of `list1`. Note that aliases should be used with caution as they can be both powerful and dangerous, as we explore more below.

Lists are an example of a *mutable* data type. They are mutable because their component parts (in this case, the elements of the list) can be modified. If you have two different variables that both refer to the same mutable piece of data – as we do with `list1` and `list2` above – changes that are made to the data's components using one of those variables will also be seen when you use the other variable, since both variables refer to the same thing.

Data types that do not allow changes to their components are termed *immutable*. Integers, floating-point values, and booleans are examples of immutable types. This isn't surprising, because they do not have any accessible component parts that could mutate anyway.

Two images here—mutation can be good ... or bad.

Types with component parts can also be immutable: for example, strings are an immutable type. If you try to change a piece of a string, Python will complain:

I love to talk. I guess that makes me immutable!

```
>>> s = 'immutable'
>>> s[0] = ' '
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignme
```

Python lets you compute what bits make up these data types, but you can't change—or even read—the individual bits themselves.

Keep in mind that, mutable or immutable, you can always use an assignment to make a variable refer to a different piece of data.

```
>>> s = 'immutable'
>>> s
'immutable'
```

```
>>> s = 'mutable'
>>> s
'mutable'
```

Takeaway Message

There are a few key ideas to keep in mind:

- All Python variables have a value (the piece of data to which they refer) and a reference (the memory location of that piece of data).
- Python assignment copies only the reference.
- Mutable data types like lists allow assignment to component parts. Immutable data types like strings do not allow assignment to component parts.
- If you have two different variables that both refer to the same mutable piece of data, changes that are made to the data's components using one of the variables will also be seen when you use the other variable.
- It is always possible to use an assignment statement to make a variable refer to a different piece of data.

5.5 Mutable Data + Iteration: Sorting out Artists

Now that we have introduced the two fundamental concepts in imperative programming, we return to our recommendation example in order to motivate and illustrate the power of data mutation and iteration together. The example we will use is sorting a list of elements.

5.5.1 Why Sort? Running Time Matters

Before we dive into the details of sorting, let's establish (at least one reason) why it is useful to have data that is in sorted order. In short, sorted data makes data processing much faster.

But how do we measure what is "fast" vs. what is "slow?" The key to analyzing how long a program takes to run is to count the number of operations that it will perform for a given size input. Computer Scientists rarely care how fast or slow programs are for very small input: they are almost always fast if the input is small. However, when processing large inputs (for example, millions of users who have each rated hundreds or even thousands of artists), speed becomes critical.

So, let's look at how long it takes to calculate the number of matches between two lists using the function from section 5.3.4, reproduced here:

```
def numMatches(list1, list2):
    ''' return the number of elements that match betwe
        list1 and list2 '''
    count = 0
    for item in list1:
        if item in list2:
            count += 1
    return count
```

In fact, if we were trying to build a system to handle millions of users, we would need to make many more optimizations and use fundamentally different algorithms, but that is beyond the scope of this chapter. Here we show you how to speed things up to handle slightly larger inputs.

Let's say for the moment that each list has four elements in it. Take a moment to think about how many comparisons you think the program will make... then read on below.

First, you take the first element of the first list, and ask if it is `in` the second list. The `in` command is a bit deceptive because it hides a significant number of comparisons. How does Python tell if an item is `in` a list? It has to compare that item to every item in the list! So in this case, the first item in the first list is compared to every item (i.e., four of them) in the second list to determine whether it is `in` that list.

"Wait!" you say. If the item is actually in the list, it doesn't actually have to check all four, and can stop checking when it finds the item in question. This is exactly correct, but in fact it doesn't matter in our analysis. For the purpose of an analysis like this, computer scientists are quite pessimistic. They rarely care what happens when things work out well–what they care about is what might possibly happen in the *worst case*. In this case, the worst case is when the item is not in the list and Python has to compare it to every item in the list to determine it's not there. Since what we care about is this worst case behavior, we will perform our analysis as if we were dealing with the worst case.

So, back to the analysis: For the first item in the list, Python made four comparisons to the items in the second list – commands that were hidden away in the `in` command. Now our program moves on to the second item in the first list, where it again makes four comparisons with the second list. Similarly, it makes four comparisons for each of the third and the fourth elements in the first list. For a total of $4 + 4 + 4 + 4 = 4 * 4 = 16$ .Again, this probably doesn't sound like such a bad number. After all, your computer can make 16 comparisons in *way* less than a second.

But what if our lists were longer? What if the user had rated 100 artists and the comparison user had rated 1000 (high, but not crazy)? Then the system would have to do 1000 comparisons (to the items in the second list) for each of the 100 items in the first list for a total of $100 * 1000 = 10^5$ comparisons. Still not huge, but hopefully you can see where this is going. In general, the matching algorithm we've written above takes $N * M$ comparisons, where N is the size of the first list and M is the size of the second list. For simplicity, we might just assume that the two lists will always be the same length, N, in which case it makes N^2 comparisons.

The good news is that we can do significantly better, but we have to make an assumption about the lists themselves. What if our lists were sorted alphabetically? How could that make our matching algorithm faster? The answer is that we can keep the lists "synchronized," so to speak, and march through both lists at the same time, rather than pulling a single element out of the first list and comparing it to all of the elements in the second. For example, if you are looking at the first element of the first list and it is "Black Eyed Peas" and the first element of the second list is "Counting Crows" you know that the Black Eyed Peas do not appear in the second list, because C is already past B. So you can simply discard the Black Eyed Peas and move on to the next artist in the first list.

Here's what the new algorithm looks like. Remember that it assumes the lists will be in sorted order (we'll talk about how to sort lists later).

Discarding the "Black Eyed Peas" would save my hearing!

- Initialize a counter to 0
- Set the current item in each list to be the first item in each list
- Repeat the following until you reach the end of one of the two lists:
- Compare the current items in each list.
- If they are equal, increment the counter and advance the current item of both lists to the next items in the lists.
- Else, if the current item in the first list is alphabetically before the current item in the second list, advance the current item in the first list.
- Else, advance the current item in the second list.
- The counter holds the number of matches.

Before you look at the code below, ask yourself: "What kind of loop should I use here? A `for` loop or `while` loop?" When you think you've got an answer you're happy with, read on.

Here's the corresponding Python code:

```
1  def numMatches( list1, list2 ):
2      '''return the number of elements that matc
3      matches = 0
4      i = 0
5      j = 0
6      while i < len(list1) and j < len(list2):
7          if list1[i] == list2[j]:
8              matches += 1
9              i += 1
10             j += 1
11         elif list1[i] < list2[j]:
12             i += 1
13         else:
14             j += 1
15     return matches
16
17 print numMatches(['a', 'l', 'i', 's', 'o', 'n'
```

<< First | < Back | Step 1 of 30 | Forward > | Last >>

line that has just executed
next line to execute

Frames Objects

(ch05_matches)

Using ==, >, <, etc. on strings is perfectly valid and these operators will compare strings alphabetically. Recall from section 4.2.3 that text is represented numerically. In this encoding all of the capital letters are mapped to consecutive numbers, with 'A' getting the lowest number and 'Z' getting the highest. The lower-case letters are also mapped to consecutive numbers, which are all higher than the number used for 'Z'. It is these encodings that are used when strings are compared. This means that strings that start with capital letters will always come "alphabetically" before those that start with lower case letters. This issue is important to consider in our program (we haven't yet), and we'll discuss it below in Section 5.5.2.

Now the question remains, is this approach really faster than the previous approach to comparing the elements in two lists? The answer is: definitely yes. Let's again look at the number of comparisons that would need to be made to compare two lists with 4 elements each. Imagine that none of the elements match and that the lists are exactly interleaved alphabetically. That is, first the element in list 1 is smaller, then the element in list 2 is smaller, and so on, as in the lists `["Amy Winehouse", "Coldplay", "Madonna", "Red Hot Chili Peppers"]` and `["Black Eyed Peas", "Dave Matthews Band", "Maroon 5", "Stevie Nicks"]`

Python 2 and earlier versions use ASCII encoding, while Python 3 uses a slightly different encoding called Unicode, but everything here applies to both encodings.

With these two lists, the code above will never trigger on the first `if` condition–it will always increment either `i` or `j`, but not both. Furthermore, it will run out of elements in one list just before it runs out of elements in the second. Essentially it will look at all the elements in both lists.

At first it might seem like we haven't made any improvements. After all, aren't we still looking at all the elements of both lists? Aha, but now there's an important difference! Whereas before we were looking at all the elements in the second list *for each element in the first list*, here we are only looking at all the elements in the second list *once*. In other words, each time through the loop, either `i` or `j` is incremented, and they are never decremented. So either `i` or `j` will hit the end of the list after all of the elements of that list have been looked at and one fewer than the elements of the other list have been looked at. So in this example, this means we will make exactly 7 comparisons. In general, if the lists are both length N , the number of comparisons this algorithm will make is $N + N - 1$ or $2N - 1$. So even for the case where one list has 100 elements and the second has 1000, this is only about 1100 comparisons, a significant improvement over the 10^5 of the previous approach!

In technical terms, computer scientists call this second algorithm a *linear time algorithm* because the equation which describes the number of comparisons is linear. The first algorithm is called a *quadratic time algorithm* because its equation is quadratic.

5.5.2 A Simple Sorting Algorithm: Selection Sort

Now that we've motivated at least one reason to sort data (there are many others—you can probably think of several of them), let's look at an algorithm for actually doing the

sorting.

Let's consider the list of artists that the user likes that we collect by prompting the user at the start of the program. Recall that the code that gathers this list is the following:

```
newPref = raw_input("Plese enter the name of an artist \
or band that you like: " )

while newPref != '':
    prefs.append(newPref)
    newPref = raw_input("Please enter an artist or band \
that you like, or just press enter to see recommendations: ")

print ('Thanks for your input!')
```

Remember that in Python 3 you'll need to use `input` rather than `raw_input`.

The artists the user has entered are stored in the list `prefs`, but that list is entirely unsorted. We would like to sort this list alphabetically, and clean up the formatting of the text while we're at it.

I don't think this will work for LMFAO!

First, to make our lives easier, and to facilitate the matching process between user profiles, we will make a pass through the list to standardize the format of the artist names. In particular, we want to be sure that there is no leading (or trailing) whitespace, and that the artist or band names are in Title Case (i.e., the first letter of each word is capitalized, and the rest are lower case). Even though this format may fail for some bands , we'll work with it because it gives us a standard representation that allows the strings to be sorted and compared without us having to worry about case issues getting in the way.

Because strings are immutable objects, we can't actually change them, but rather we have to generate new strings with the formatting we desire. Here is the code that accomplishes this goal (note that we also build a new list, which is not strictly necessary):

```
standardPrefs = []
for artist in prefs:
    standardPrefs.append(artist.strip().title())
```

Recall that all uppercase strings are considered "less than" lower case strings, and it would be confusing to the user to alphabetically sort ZZ Top before adele.

The `strip` function returns a new string identical to `artist`, but

without any leading or trailing whitespace. Then the `title` function returns that same string in Title Case.

Now that our data is standardized, we can sort it from smallest (alphabetically first) to largest. Sorting is an important topic of study in computer science in its own right.

Here we will discuss one sorting algorithm, but there's much more to be said on the subject.

I sort of knew that it was important!

To develop our algorithm, we start with small cases: what are the computational pieces that enable the rearrangement of a list?

A two-element list is the smallest (non-trivial) case to consider: in the worst case, the two elements would need to swap places with each other.

In fact, this idea of a `swap` is all we need! Imagine a large list that we'd like sorted in ascending order.

- First, we could find the smallest element. Then, we could swap that smallest element with the first element of the list.
- Next, we would search for the second-smallest element, which is the smallest element of the rest of the list. Then, we could swap it into place as the second element of the overall list.
- We continue this swapping so that the third-smallest element takes the third place in the list, do the same for the fourth, ... and so on ... until we run out of elements.

This algorithm is called *selection sort* because it proceeds by repeatedly selecting the minimum remaining element and moving it to the next position in the list.

What functions will we need to write selection sort? It seems we need only two:

- `index_of_min(aList,starting_index)`, which will return the index of the smallest element in `aList`, starting from index `starting_index`.
- `swap(aList,i,j)`, which will swap the values of `aList[i]` and `aList[j]`

Here is the Python code for this algorithm:

```
1  def selectionSort(listToSort):
2      '''sorts aList iteratively and in-place'''
3      for starting_index in range(len(listToSort
4          min_elem_index = index_of_min(listToSo
5          swap(listToSort, starting_index, min_e
6      return listToSort
7
8  # And here is index_of_min!:
9  def index_of_min(aList, start_index):
10     '''returns the index of the min element at
11     min_elem_index = start_index
12     for i in range(start_index, len(aList)):
13         if aList[i] < aList[min_elem_index]:
14             min_elem_index = i
15     return min_elem_index
16
17 #And swap:
```

<< First | < Back | Step 1 of 55 | Forward > | Last >>

line that has just executed
next line to execute

Frames Objects

(ch05_selSort)

Notice that the code above does not return anything. But when we run it, we see that it works. After the call to `selectionSort`, our list is sorted!

```
>>> standardPrefs
['Maroon 5', 'Adele', 'Lady Gaga']
>>> selectionSort(standardPrefs)
>>> standardPrefs
['Adele', 'Lady Gaga', 'Maroon 5']
```

5.5.3 Why `selectionSort` Works

Why does this code work? What is more, why does it work *even though* `swap` *does not*

have a `return` *statement*? There are two key factors.

First, lists are mutable. Thus, two (or more) variables can refer to the same list, and changes that are made to list elements using one variable will also be seen when you use the other variables. But where are the two variables referring to the same list? It seems that `standardPrefs` is the only variable referring to the original three-element list in the example above.

This is the second factor: when inputs are passed into functions, the function parameters are assigned to each input just as if an assignment statement had been explicitly written, e.g.,

```
listToSort = standardPrefs
```

occurs at the beginning of the call to `selectionSort(standardPrefs)`.

As a result, as long as `listToSort` and `standardPrefs` refer to the same list, any changes that are made to `listToSort`'s elements will affect `standardPrefs`'s elements —because `listToSort`'s elements and `standardPrefs`'s elements are one and the same thing.

Takeaway message: *Passing inputs into a function is equivalent to assigning those inputs to the parameters of that function. Thus, the subtleties of mutable and immutable data types apply, just as they do in ordinary assignment.*

The left side of Figure 5.4 illustrates what is happening at the beginning of the first call to `swap`. `swap`'s variables are displayed in red, while `selectionSort`'s variables are displayed in black. Notice that `swap`'s `i` and `j` refer to the index locations in the list `aList` (which is just another name for the list `listToSort`).

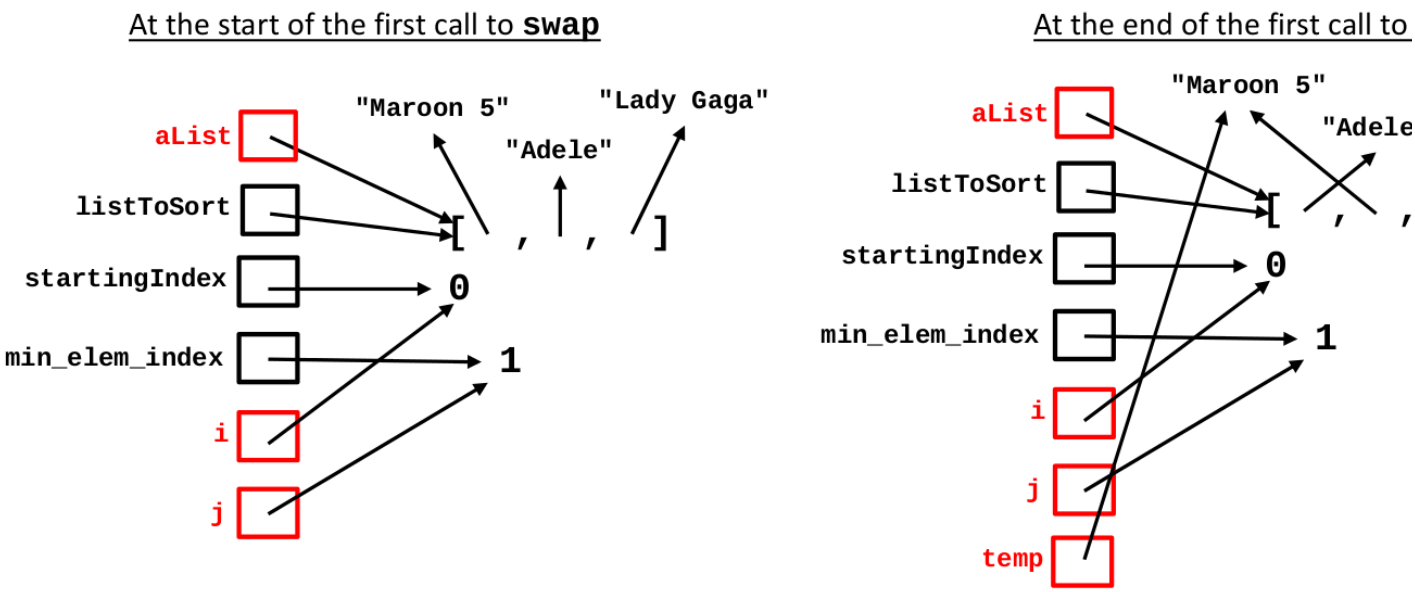


Figure 5.4: A graphical depiction of the variables in `selectionSort` and `swap` at the beginning and the end of the first call to `swap`

The right side of Figure 5.4 shows how the variables look at the very end of the first call to `swap`. When swap finishes, its variables disappear. But notice that even when `swap`'s `aList`, `i`, `j`, and `temp` are gone, the value of the list has been changed in memory. Because `swap`'s `aList` and `selectionSort`'s `listToSort` and the original `standardPrefs` are all references to the same collection of mutable elements, the effect of assignment statements on those elements will be shared among all of these names. After all, they all name the same list!

5.5.4 A Swap of a Different Sort

Consider a very minor modification to the `swap` and `selectionSort` functions that has a very major impact on the results:

```
def selectionSort(listToSort):
    '''sorts listToSort iteratively and in-place'''
    for starting_index in range(len(listToSort)):
        min_elem_index = index_of_min(listToSort, starting_index)
        # now swap the elements!
        swap(listToSort[starting_index], listToSort[min_elem_index])

def swap(a, b):
    '''swaps the values of a and b'''
    temp = a
    a = b
    b = temp
```

The code looks almost the same, but now it does not work!

```
>>> standardPrefs     # Assume this list is already populated with standa
['Maroon 5', 'Adele', 'Lady Gaga']
>>> selectionSort(standardPrefs)
>>> standardPrefs
['Maroon 5', 'Adele', 'Lady Gaga']
```

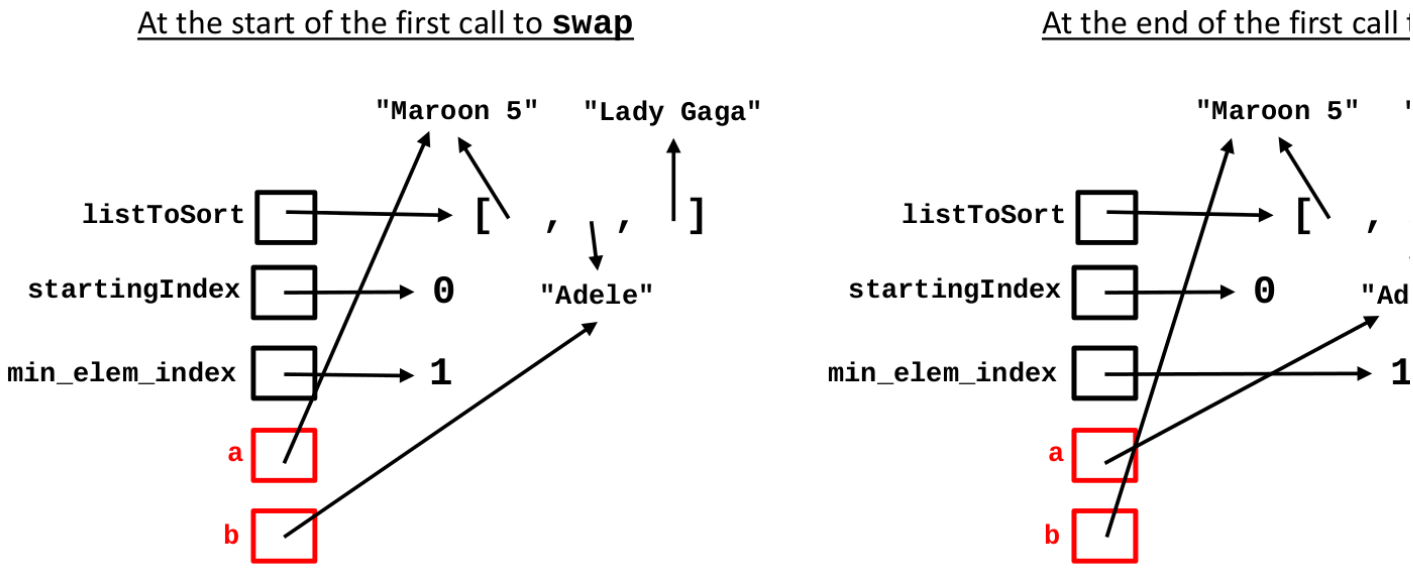


Figure 5.5: A graphical depiction of the variables in `selectionSort` and the new (and faulty) `swap` at the beginning and the end of the first call to `swap`

The variables `a` and `b` in `swap` indeed do get swapped, as the following before-and-after figure illustrates. But nothing happens to the elements of `aList` or, to say the same thing, nothing happens to the elements of `standardPrefs`.

Figure 5.5 illustrates what is happening before and after the call to `swap`.

What happened? This time, `swap` has only two parameters, `a` and `b`, whose references are copies of the references of `listToSort[start_index]` and `listToSort[min_elem_index]`, respectively. Keep in mind Python's mechanism for assignment: *Assignments make a copy of a reference.*

Thus, when `swap` runs this time, its assignment statements are working on *copies* of references. All of the swapping happens just as specified, so that the values referred to by `a` and `b` are, indeed, reversed. But the references held within `selectionSort`'s list `listToSort` have not changed. Thus, the value of `listToSort` does not change. Since the value of `standardPrefs` *is* the value of `listToSort`, nothing happens.

As we mentioned at the start of this section, Selection Sort is just one of many sorting algorithms, some of which are faster than others (Selection Sort is not particularly fast, though it's certainly not the slowest approach either). In practice, especially for now, you'll probably just call Python's built-in `sort` function for lists, which is efficient, and more importantly, correctly implemented.

```
>>> standardPrefs     # Assume this list is already populated with standa
['Maroon 5', 'Adele', 'Lady Gaga']
>>> standardPrefs.sort()
>>> standardPrefs
['Adele', 'Lady Gaga', 'Maroon 5']
```

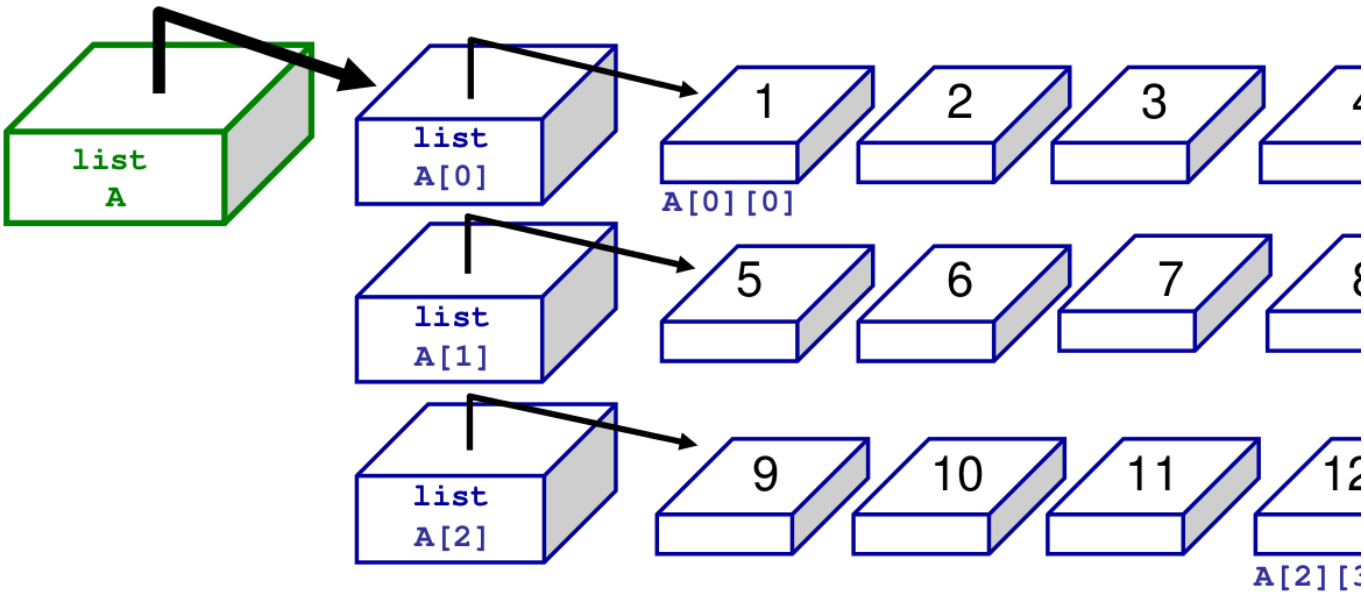


Figure 5.6: A graphical depiction of a 2D array. Following our model from above, the green box is stored in the CPU, and the rest of the data (including the references to other lists) is stored in memory.

5.5.5 2D Arrays and Nested Loops

It turns out that you can store more than just numbers in lists: arbitrary data can be stored. We've already seen many examples in Chapter 2 where lists were used to store strings, numbers, and combinations of those data types. Lists of lists are not only possible, but powerful and common.

In an imperative context, lists are often called *arrays*, and in this section we examine

another common array structure: arrays that store other arrays, often called *2D arrays*.

The concept behind a 2D array is simple: it is just a list whose elements themselves are lists. For example, the code

```
>>> a2DList = [[1, 2, 3, 4],[5, 6, 7, 8], [9, 10, 11,
```

creates a list named `a2DList` where each of `a2DList`'s elements is itself a list. Figure 5.6 illustrates this 2D array (list) graphically. In the figure the array name has been shortened to just `A` for conciseness.

Our attorneys have advised us to note that technically, there are very subtle differences between "lists" and "arrays" – but the distinctions are too fine to worry about here.

You can access the individual elements in the 2D array by using "nested" indices as shown below:

```
>>> a2DList[0]    # Get the first element of a2DList
[1, 2, 3, 4]
>>> a2DList[0][0] # Get the first element of a2DList[0]
1
>>> a2DList[1][1] # Get the second element of a2DList[1]
6
```

We can also ask questions about `a2DList`'s height (the number of rows) and its width (the number of columns):

```
>>> len(a2DList)    # The number of elements (rows) in a2DList
3
>>> len(a2DList[0]) # The number of elements in a list in a2DList (i.e.,
4
```

Typically, arrays have equal-length rows, though "jagged" arrays are certainly possible.

This 2D structure turns out to be a very powerful tool for representing data. In fact, we've already seen an example of this in our recommender program: the stored users' preferences were represented by a list-of-lists, i.e., a 2D array. In this case, the array certainly was "jagged" as different stored users had rated different numbers of artists.

Above we wrote some code to standardize the representation of one particular user's preferences. But we might be unsure about the database—are those lists also standardized (and sorted)? If not, no problem. It's easy to write code that quickly

standardizes all the elements in all of the lists:

```
1  def standardizeAll(storedPrefs):
2      ''' Returns a new list of lists of stored
3      with each artist string in Title Case,
4      with leading and trailing whitespace remov
5      '''
6      standardStoredPrefs = []
7      for storedUser in storedPrefs:
8          standardStoredUser = []
9          for artist in storedUser:
10             standardStoredUser.append(artist.s
11         standardStoredPrefs.append(standardSto
12     return standardStoredPrefs
13
14 print (standardizeAll([[' adele', 'lAdY GAGA']
```

<< First | < Back | Step 1 of 20 | Forward > | Last >>

line that has just executed

next line to execute

Frames Objects

(ch05_stand)

In the above code, the outer loop controls the iteration over the individual users. That is, for each iteration of the outer loop, the variable `storedUser` is assigned a list containing the preferences of one of the users. The inner loop then iterates over the elements in that user's list of preferences. In other words, `artist` is simply a string.

We can also write the above code as follows:

```
def standardizeAll(storedPrefs):
    ''' Returns a new list of lists of stored user preferences,
        with each artist string in Title Case,
        with leading and trailing whitespace removed.
    '''
    standardStoredPrefs = []
    for i in range(len(storedPrefs)):
        standardStoredUser = []
```

```
            for j in range(len(storedPrefs[i])):
                standardStoredUser.append(storedPrefs[i][j].strip().title())
            standardStoredPrefs.append(standardStoredUser)
        return standardStoredPrefs
```

This code does the same thing, but we use the indexed version of the `for` loop instead of letting the `for` loop iterate directly over the elements in the list. Either construct is fine, but the latter makes the next example more clear.

Because lists are mutable, we don't actually have to create a whole new 2D array just to change the formatting of the strings in the array. Remember that strings themselves are not mutable, so we can't directly change the strings stored in the list. However, we can change *which strings* are stored in the original list, as follows:

```
def standardizeAll(storedPrefs):
    ''' Mutates storedPrefs so that each string is in
        Title Case, with leading and trailing whitespace removed.
        Returns nothing.
    '''
    for i in range(len(storedPrefs):
        for j in range(len(storedPrefs[i])):
            standardArtist = storedPrefs[i][j].strip().title()
            storedPrefs[i][j] = standardArtist
```

Notice that this code is slightly simpler than the code above, and avoids the overhead of creating a whole new list of lists.

5.5.6 Dictionaries

So far we've looked at one type of mutable data: arrays. Of course, there are many others. In fact, in the next chapter you'll learn how to create your own mutable data types. But more on that later. For now we will examine a built-in datatype called a *dictionary* that allows us to create mappings between pieces of data.

Have you ever looked up `dictionary` *in a*

To motivate the need for a dictionary, let's once again return to the recommender program. Until now, in our examples we have not been associating stored user's names with their preferences. While we don't need to store the user's name with their examples in order to calculate the best matching user (assuming we know that the list of stored users does not contain the current user!), there are many advantages of doing so. For one, we can make

sure that the user doesn't get matched against themselves! *dictionary?*
Additionally, it might be nice in an extended version of the system to suggest possible "music friends" to the user, whose tastes they may want to track.

So how will we associate user names with their preferences? One way would be to simply use lists. For example, we could have a convention where the first element in each user's stored preferences is actually the name of the user herself. This approach works, but it is not very elegant and is likely to lead to mistakes. For example, what if another programmer working on this system forgets that this is the representation and starts treating user names as artist names? Perhaps not a tragedy, but incorrect behavior nonetheless. In short, elegant design is important!

What we need is a single place to store the mapping from user names to user preferences that we can pass around to all of the functions that need to know about it. This is where the dictionary comes in handy! A dictionary allows you to create mappings between pieces of (immutable) data (the *keys*) and other pieces of (any kind of) data (the *values*). Here's an example of how they work:

```
>>> myDict = {}     # creates a new empty dictionary

# create an association between 'April' her
# list of preferred artists.
# 'April' is the key and the list is the value
>>> myDict['April'] = ['Maroon 5', 'The Rolling Stones',
                       'The Beatles']

# Ditto for Ben and his list
>>> myDict['Ben'] =  ['Lady Gaga', 'Adele',
                      'Kelly Clarkson', 'The Dixie Chicks']
>>> myDict                  # display the mappings currently in the dicti
{'April': ['Maroon 5', 'The Rolling Stones', 'The Beatles'],
'Ben': ['Lady Gaga', 'Adele', 'Kelly Clarkson', 'The Dixie Chicks']}
>>> myDict['April']         # get the item mapped to 'April'
['Maroon 5', 'The Rolling Stones', 'The Beatles']
>>> myDict['f']             # get the item mapped to 'f'. Will cause an e
                            # mappings are one-way--'f' is not
                            # a valid key.
Traceback (most recent call last):
File "<pyshell\#14>", line 1, in <module>
myDict['f']
KeyError: 'f'
>>> myDict.has_key('f')     # Check whether a key is in the
                            # dictionary
False
>>> myDict.keys()           # Get the keys in the dictionary
[April', 'Ben']             # Notice that this is a special kind of
                            # object that you can iterate
```

```
                            # over, but that's not a list
>>> myDict[1] = 'one'       # keys can be any immutable type
>>> myDict[1.5] = [3, 5, 7] # values can also be mutable, and
                            # we can mix types in the same
                            # dictionary
>>> myDict[[1, 2]] = 'one'  # Keys cannot be mutable

Traceback (most recent call last):
 File "<pyshell\#36>", line 1, in <module>
 myDict[[1, 2]] = 'one'
TypeError: list objects are unhashable

# a shorthand way to create a dictionary
>>> userPrefs = {'April': ['Maroon 5', 'The Rolling Stones', 'The Beatles
'Ben': ['Lady Gaga', 'Adele', 'Kelly Clarkson', 'The Dixie Chicks']}
>>> userPrefs
'April': ['Maroon 5', 'The Rolling Stones', 'The Beatles'],
'Ben': ['Lady Gaga', 'Adele', 'Kelly Clarkson', 'The Dixie Chicks']
```

The order in which the key-value pairs appear in Python's output of `userPrefs` represents its internal representation of the data. They are not always in the same order.

Now let's look at how we can modify our recommender code to use a dictionary instead of a list of lists:

```
def getBestUser(currUser, prefs, userMap):
    ''' Gets recommendations for currUser based on the users in userMap (
    and the current user's preferences in prefs (a list) '''
    users = userMap.keys()
    bestuser = None
    bestscore = -1
    for user in users:
        score = numMatches(prefs, userMap[user])
        if score > bestscore and currUser != user:
            bestscore = score
            bestuser = user
    return bestuser
```

Notice that dictionaries allow us to make sure we're not matching a user to their own stored preferences! Pretty simple... and flexible too!

5.6 Reading and Writing Files

We've almost got all the pieces we need to build our recommender program. The last major piece we are missing is the ability to read and write files, which we will need to load

and store the preferences for the users the system already knows about.

Fortunately, working with files is very simple in Python, and we illustrate file input and output (i.e., file I/O) through example. Imagine that our stored user preferences exist in a file named `musicrec-store.txt`. There is one line in the file for each user and the format of each line is the following:

```
username:artist1,artist2,...,artistN
```

We can write the following code to read in and process all the lines from this file:

```
def loadUsers(fileName):
    ''' Reads in a file of stored users' preferences stored
        in the file 'fileName'.
        Returns a dictionary containing a mapping of user
        names to a list of preferred artists
    '''
    file = open(fileName, 'r')
    userDict = {}
    for line in file:
        # Read and parse a single line
        [userName, bands] = line.strip().split(":")
        bandList = bands.split(",")
        bandList.sort()
        userDict[userName] = bandList
    file.close()
    return userDict
```

When we call this function we would pass it the filename `musicrec-store.txt`.

There are a few key pieces to this function. First, we have to open the file for reading. This task is accomplished with the line `file = open(fileName, 'r')`, which gives us a link to the contents of the file through `file`, but we can only read from this file (we cannot write to it). If we wanted to write to the file, we would specify `'w'` as the second argument to the `open` function; if we wanted to both read and write, we would pass `'w+'` as the second argument.

Once we have our handle to the file's contents (`file`) we can read lines from it using the `for` loop construct above. As the `for` loop iterates over `file`, what it is actually doing is pulling out one line at a time until there are no more lines, at which point the `for` loop ends.

Finally, when we have read all the data out of the file and stored it in the dictionary in our program, we can close the file using `file.close()`.

The function below shows the part of the code that writes the stored user preferences (including the current user) back to the file:

```
def saveUserPreferences(userName, prefs, userMap, fileName):
    ''' Writes all of the user preferences to the file.
        Returns nothing. '''
    userMap[userName] = prefs
    file = open(fileName, "w")
    for user in userMap:
        toSave = str(user) + ":" + ",".join(userMap[user]) + \
                 "\n"
        file.write(toSave)
    file.close()
```

5.7 Putting It All Together: Program Design

Now that we've got all the tools, it's time to construct the whole recommender program. However, as a disclaimer, large scale program design can at times feel more like an art than a science. Maybe more science than art! There are certainly guiding principles and theories, but you rarely "just get it right" on your first try, no matter how careful you are. So expect that in implementing any program of reasonable size, you'll need to make a couple of revisions. Think of it like writing a paper—if you like writing papers.

CS is at least as creative and precise as careful writing; don't be afraid of creating—and redoing—several drafts!

The first step to program design is trying to figure out what data your program is responsible for and how that data comes into the program (input), gets manipulated (computation) and is output from the program (output). In fact, this task is so important that we had you do this at the beginning of the chapter.

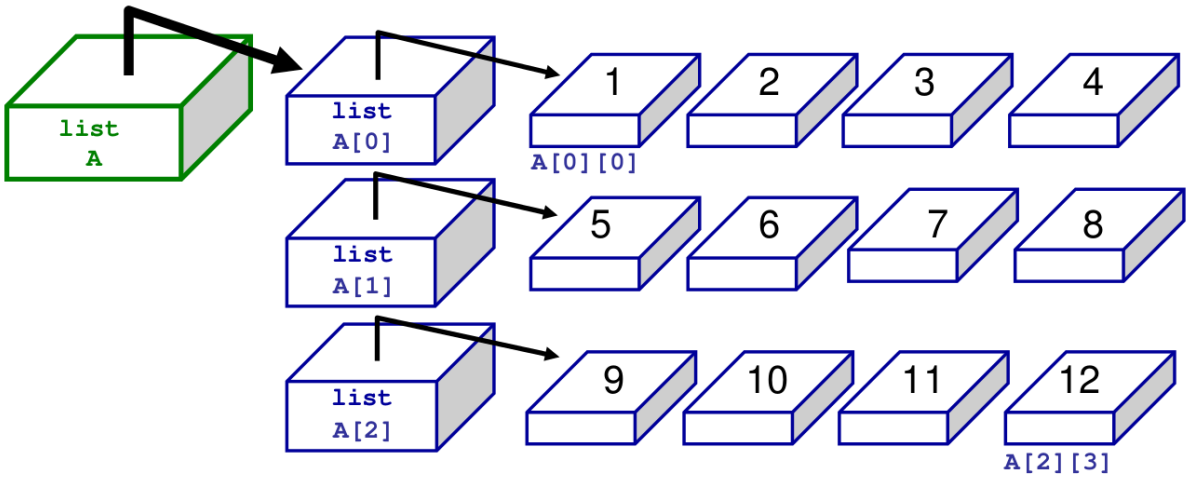
After identifying the data your program must keep track of, the next step is to decide what data structures you will use to keep track of this data. Do you need ints, lists, or dictionaries, or some other structure entirely? Take a moment to choose variable names and data types for each of the piece of data you identified at the beginning of the chapter (or that you identify now).

Finally, now that you have identified our data and the computation that our program needs to perform, you are ready to start writing functions. We can start with any function

we like–there's no right order as long as we are able to test each function as we write it.

Our complete program is given below in listing 5.1. Let's return to the idea of collaborative filtering and examine how this program implements the basic CF algorithm we described in section 5.1. As we described there, the basic idea behind collaborative filtering is to try to make predictions about a user by looking at data from a number of other users. There are two basic steps to a CF algorithm:

- Find the user (or users) most similar to the current user,
- Make predictions based on what they like.



In the code in listing 5.1, these two steps are performed in helper functions called in `getRecommendations`. First, `findBestUser` finds and returns the user whose tastes are closest to the current user. Then `drop` returns a list of the artists who the best user likes who are not already in the current user's list.

Notice that even though the code is relatively short, we have chosen to separate the two stages of the algorithm into two separate functions for clarity. Generally it is a good idea for each semantic piece of your algorithm to have its own function. Notice also that `findBestUser` also relies on a helper function (`numMatches`), which again helps make the specific functionality of each piece of the code more clear.

One more item that's new in the code below is the last line:

```
if __name__ == "__main__": main()
```

You will notice that the function `main` implements the main control of the recommendation

program. We could run our recommendation program by loading this code into the Python interpreter and then typing:

```
>>> main()
```

However, to keep the user from having to type this extra command, we include the above line, which tells the program to automatically run the function `main` as soon as the code is loaded into the interpreter.

Finally, you'll notice the variable `PREF_FILE` at the top of the program. This variable is known as a *global variable* because it is accessible to all of the functions in the program. By convention, global variable names are often ALL CAPS to distinguish them from local variables and parameters. Generally global variables should be avoided, but there are a few special cases such as this one where we want to avoid sprinkling a hard-coded value (in this case, the name of the file) throughout the program. Using a global variable for the filename makes it easy to change in the future.

```
# A very simple music recommender system.

PREF_FILE = "musicrec-store.txt"

def loadUsers(fileName):
    ''' Reads in a file of stored users' preferences
        stored in the file 'fileName'.
        Returns a dictionary containing a mapping
        of user names to a list preferred artists
    '''
    file = open(fileName, 'r')
    userDict = {}
    for line in file:
        # Read and parse a single line
        [userName, bands] = line.strip().split(":")
        bandList = bands.split(",")
        bandList.sort()
        userDict[userName] = bandList
    file.close()
    return userDict

def getPreferences(userName, userMap):
    ''' Returns a list of the uesr's preferred artists.

        If the system already knows about the user,
        it gets the preferences out of the userMap
        dictionary and then asks the user if she has
        additional preferences.  If the user is new,
        it simply asks the user for her preferences. '''
```

```
    newPref = ""
    if userName in userMap:
        prefs = userMap[userName]
        print("I see that you have used the system before.")
        print("Your music preferences include:")
        for artist in prefs:
            print(artist)
        print("Please enter another artist or band that you")
        print("like, or just press enter")
        newPref = input("to see your recommendations: ")
    else:
        prefs = []
        print("I see that you are a new user.")
        print("Please enter the name of an artist or band")
        newPref = input("that you like: " )

    while newPref != "":
        prefs.append(newPref.strip().title())
        print("Please enter another artist or band that you")
        print("like, or just press enter")
        newPref = input("to see your recommendations: ")

    # Always keep the lists in sorted order for ease of
    # comparison
    prefs.sort()
    return prefs


def getRecommendations(currUser, prefs, userMap):
    ''' Gets recommendations for a user (currUser) based
        on the users in userMap (a dictionary)
        and the user's preferences in pref (a list).
        Returns a list of recommended artists.  '''
    bestUser = findBestUser(currUser, prefs, userMap)
    recommendations = drop(prefs, userMap[bestUser])
    return recommendations

def findBestUser(currUser, prefs, userMap):
    ''' Find the user whose tastes are closest to the current
        user.  Return the best user's name (a string) '''
    users = userMap.keys()
    bestUser = None
    bestScore = -1
    for user in users:
        score = numMatches(prefs, userMap[user])
        if score > bestScore and currUser != user:
            bestScore = score
            bestUser = user
    return bestUser

def drop(list1, list2):
    ''' Return a new list that contains only the elements in
        list2 that were NOT in list1. '''
    list3 = []
```

```
    i = 0
    j = 0
    while i < len(list1) and j < len(list2):
        if list1[i] == list2[j]:
            i += 1
            j += 1
        elif list1[i] < list2[j]:
            i += 1
        else:
            list3.append(list2[j])
            j += 1

    return list3

def numMatches( list1, list2 ):
    ''' return the number of elements that match between
        two sorted lists '''
    matches = 0
    i = 0
    j = 0
    while i < len(list1) and j < len(list2):
        if list1[i] == list2[j]:
            matches += 1
            i += 1
            j += 1
        elif list1[i] < list2[j]:
            i += 1
        else:
            j += 1
    return matches

def saveUserPreferences(userName, prefs, userMap, fileName):
    ''' Writes all of the user preferences to the file.
        Returns nothing. '''
    userMap[userName] = prefs
    file = open(fileName, "w")
    for user in userMap:
        toSave = str(user) + ":" + ",".join(userMap[user]) + \
                 "\n"
        file.write(toSave)
    file.close()

def main():
    ''' The main recommendation function '''
    userMap = loadUsers(PREF_FILE)
    print("Welcome to the music recommender system!")

    userName = input("Please enter your name: ")
    print ("Welcome,", userName)

    prefs = getPreferences(userName, userMap)
    recs = getRecommendations(userName, prefs, userMap)

    # Print the user's recommendations
```

```
    if len(recs) == 0:
        print("I'm sorry but I have no recommendations")
        print("for you right now.")
    else:
        print(userName, "based on the users I currently")
        print("know about, I believe you might like:")
        for artist in recs:
            print(artist)

        print("I hope you enjoy them! I will save your")
        print("preferred artists and have new")
        print(" recommendations for you in the future")

    saveUserPreferences(userName, prefs, userMap, PREF_FILE)


if __name__ == "__main__": main()
```

5.8 Conclusion

We've covered a lot of ground in this chapter, ultimately resulting in a recommender program similar in spirit to ones that you undoubtedly use yourself.

Are there any recommender programs that recommend recommender programs?

We've also seen that there are often multiple different ways of doing the same thing. For example, recursion, `for` loops, and `while` loops all allow us to repeat a computational task. How do you determine which one to use? Some problems are inherently recursive. For example, the edit distance problem from Chapter 2 was very naturally suited for recursion because it can be solved using solutions to smaller versions of the same problem – the so-called "recursive substructure" problem. Other problems, like computing the factorial of a number, can be solved naturally – albeit differently – using recursion or loops. And then there are cases where loops seem like the best way to do business. The fact that there are choices like these to be made is part of what makes designing programs fun and challenging.

Chapter 6: Fun and Games with OOPs: Object-Oriented Programs

> *I paint objects as I think them, not as I see them.*
>
> —Pablo Picasso

6.1 Introduction

In this chapter we'll develop our own "killer app": A 3D video game called "Robot versus Zombies." By the end of this chapter you'll have the tools to develop all kinds of interactive 3D programs for games, scientific simulations, or whatever else you can imagine.

We're sneaky! The *true* objective of this chapter is to demonstrate a beautiful and fundamental concept called *object-oriented programming*. Object-oriented programming is not only the secret sauce in 3D graphics and video games, it's widely used in most modern large-scale software projects. In this chapter, you'll learn about some of the fundamental ideas in object-oriented programming. And, yes, we'll write that video game too!

6.2 Thinking Objectively

Before we get to the Robot versus Zombies video game, let's imagine the following scenario. You're a summer intern at Lunatix Games, a major video game developer. One of their popular games, Lunatix Lander, has the player try to land a spaceship on the surface of a planet. This requires the player to fire thrusters to align the spaceship with the landing site and slow it down to a reasonable landing speed. The game shows the player how much fuel remains and how much fuel is required for certain maneuvers.

As you test the game, you notice that it often reports that there is not enough fuel to perform a key maneuver when, in fact, you are certain there should be just the right amount of fuel. Your task is to figure out what's wrong and find a way to fix it.

Each of the rocket's two fuel tanks has a capacity of 1000 units; the fuel gauge for each tank reports a value between 0 and 1.0, indicating the fraction of the capacity remaining

in that tank. Here is an example of one of your tests, where `fuelNeeded` represents the fraction of a 1000 unit tank required to perform the maneuver, and `tank1` and `tank2` indicate the fraction of the capacity of each of the two tanks. The last statement is checking to see if the total amount of fuel in the two tanks equals or exceeds the fuel needed for the maneuver.

```
>>> fuelNeeded = 42.0/1000
>>> tank1 = 36.0/1000
>>> tank2 = 6.0/1000
>>> tank1 + tank2 >= fuelNeeded
False
```

Notice that $\frac{36}{1000} + \frac{6}{1000} = \frac{42}{1000}$ and the fuel needed is exactly $\frac{42}{1000}$. Strangely though, the code reports that there is not enough fuel to perform the maneuver, dooming the ship to crash.

Bummer!

When you print the values of `fuelNeeded`, `tank1`, `tank2`, and `tank1 + tank2` you see the problem:

```
>>> fuelNeeded
0.042000000000000003
>>> tank1
0.035999999999999997
>>> tank2
0.0060000000000000001
>>> tank1 + tank2
0.041999999999999996
```

This example of *numerical imprecision* is the result of the inherent error that arises when computers try to convert fractions into floating-point numbers (numbers with a decimal-point representation). However, assuming that all of the quantities that you measure on your rocket are always rational numbers—that is fractions with integer numerators and denominators—this imprecision problem can be avoided! How? Integers don't suffer from imprecision. So, for each rational number, we can store its integer numerator and denominator and then do all of our arithmetic with integers.

For example, the rational number $\frac{36}{1000}$ can be stored as the pair

To be precise, imprecision occurs because computers use only a fixed number of bits of data to represent data. Therefore, only a finite number of different quantities can be stored. In particular, the fractional part of a floating-point number, the mantissa, *must be rounded to the nearest one of the finite number of values that the*

of integers 36 and 1000 rather than converting it into a floating-point number. To compute $\frac{36}{1000} + \frac{6}{1000}$ we can compute $36 + 6 = 42$ as the numerator and 1000 as the denominator. Comparing this to the `fuelNeeded` value of $\frac{42}{1000}$ involves separately comparing the numerators and denominators, which involves comparing integers and is thus free from numerical imprecision.

computer can store, resulting in the kinds of unexpected behavior that we see here.

A rational thing to have!

Or even a fraction *of them all.*

So, it would be great if Python had a way of dealing with rational numbers as pairs of integers. That is, we would like to have a "rational numbers" type of data (or *data type*, as computer scientists like to call it) just as Python has an integer data type, a string data type, and a list data type (among others). Moreover, it would be great if we could do arithmetic and comparisons of these rational numbers just as easily as we can do with integers.

The designers of Python couldn't possibly predict all of the different data types that one might want. Instead, Python (like many other languages) has a nice way to let you, the programmer, define your own new types and then use them nearly as easily as you use the built-in types such as integers, strings, and lists.

This facility to define new types of data is called *object-oriented programming* or OOP and is the topic of this chapter.

6.3 The Rational Solution

Let's get started by defining a rational number type. To do this, we build a Python "factory" for constructing rational numbers. This factory is called a *class* and it looks like this:

```
class Rational:
    def __init__(self, num, denom):
        self.numerator = num
        self.denominator = denom
```

Don't worry about the weird syntax; we'll come back to that in a moment when we take a closer look at the details. For now, the big idea is that once we've written this `Rational`

class (and saved it in a file with the same name but with the suffix `.py` at the end, in this case `Rational.py`) we can "manufacture" – or more technically *instantiate* – new rational numbers to our heart's content. Here's an example of calling this "factory" to instantiate two rational numbers $\frac{36}{1000}$ and $\frac{6}{1000}$:

```
r1 = Rational(36, 1000)
r2 = Rational(6, 1000)
```

What's going on here? When Python sees the instruction

```
r1 = Rational(36, 1000)
```

it does two things. First, it instantiates an empty object which we'll call `self`. Actually, `self` is a *reference* to this empty object as shown in Figure 6.1.

Perhaps this is `self` *-ish of Python.*

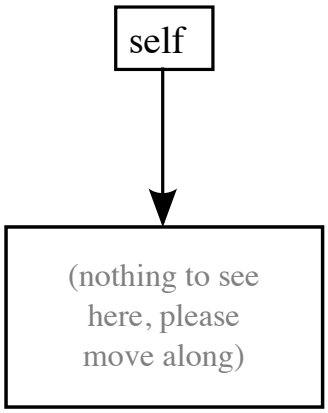


Figure 6.1: `self` refers to a new empty object. It's only empty for a moment!

Next, Python looks through the `Rational` class definition for a function named `__init__` (notice that there are two underscore characters before and after the word "init"). That's a funny name, but it's a convention in Python. Notice that in the definition of `__init__` above, that function seems to take *three* arguments, but the line `r1 = Rational(36, 1000)` has only supplied *two*. That's weird, but—as you may have guessed—the first argument is passed in automatically by Python and is a reference to the new empty `self` object that we've just had instantiated for us.

The `__init__` function takes the reference to our new empty object called `self`, and it's going to add some data to that object. The values 36 and 1000 are passed in to `__init__` as `num` and `denom`, respectively. Now, when the `__init__` function executes

the line `self.numerator = num`, it says, "go into the object referenced by `self` , give it a variable called `numerator`, and give that variable the value that was passed in as `num`." Similarly, the line `self.denominator = denom` says "go into the object referenced by `self`, give it a variable called `denominator`, and give that variable the value that was passed in as `denom`." Note that the the names `num`, `denom`, `numerator`, and `denominator` are not special—they are just the names that we chose.

The variables `numerator` and `denominator` in the `__init__` function are called `attributes` of the `Rational` class. A class can have as many attributes as you wish to define for it. It's pretty clear that a rational number class would have to have at least these two attributes!

Python uses the name "attributes" for the variables that belong to a class. Some other languages use names like "data members", "properties", "fields", and "instance variables" for the same idea.

The last thing that happens in the line

```
r1 = Rational(36, 1000)
```

is that the variable `r1` is now assigned to be a reference to the object that Python just created and we initialized. We can see the contents of the rational numbers as follows:

```
>>> r1.numerator
36
```

We used the "dot" in the function `__init__` as well, when we said `self.numerator = num`. The "dot" was doing the same thing there. It said, "go into the `self` object and look at the attribute named `numerator`." Figure 6.2 shows the situation now.

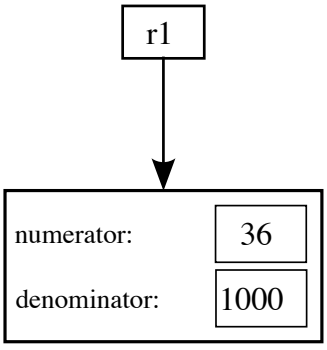


Figure 6.2: `r1` refers to the `Rational` object with its `numerator` and `denominator`.

We note that in our figures in this chapter, we're representing memory in a somewhat

different (and simpler) way than we used in Chapter 5. For example, Figure 6.2 shows the values of numerator and denominator as if they were stored inside the variables. In reality, as we saw in Chapter 5, the integer values would be somewhere else in memory, and the variables would store references to those values.

In our earlier example, we "called" the `Rational` "factory" twice to instantiate two different rational numbers in the example below:

```
r1 = Rational(36, 1000)
r2 = Rational(6, 1000)
```

The first call, `Rational(36, 1000)` instantiated a rational number, `self`, with `numerator` 36 and `denominator` 1000. This was called `self`, but then we assigned `r1` to refer to this object. Similarly, the line `r2.numerator` is saying, "go to the object called `r2` and look at its attribute named `numerator`." It's important to keep in mind that since `r1` and `r2` are referring to two different objects, each one has its "personal" `numerator` and `denominator`. This is shown in Figure 6.3.

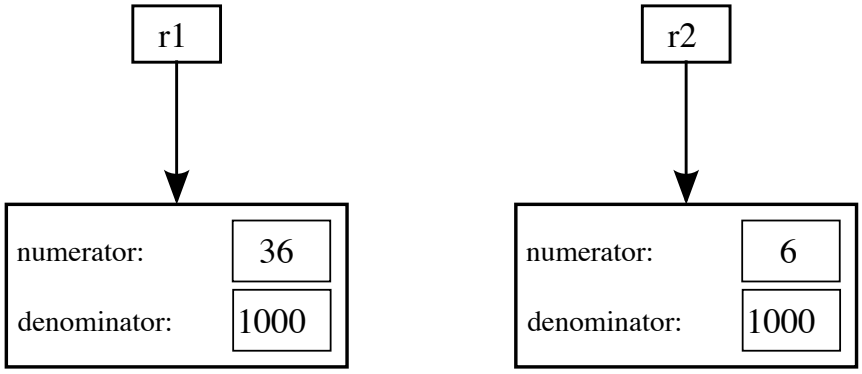


Figure 6.3: Two `Rational` numbers with references `r1` and `r2`.

Let's take stock of what we've just seen. First, we defined a factory—technically known as a *class*–called `Rational`. This `Rational` class describes a template for manufacturing a new type of data. That factory can now be used to instantiate a multitude of items—technically known as *objects*–of that type. Each object will have its own variables – or "attributes" – in this case `numerator` and `denominator`, each with their own values.

This would be a short chapter if that was the whole story!

That's cute, but is that it? Remember that our motivation for defining the `Rational` numbers was to have a way to manipulate (add, compare, etc.) rational numbers without having to convert them to the floating-point world where numerical imprecision can cause

headaches (and rocket failures, and worse).

Python's built-in data types (such as integers, floating-point numbers, and strings) have the ability to be added, compared for equality, etc. We'd like our `Rational` numbers to have these abilities too! We'll begin by adding a function to the `Rational` class that will allow us to add one Rational to another and return the sum, which will be a `Rational` as well. A function defined inside a class has a special fancy name—it's called a *method* of that class. The `__init__` method is known as the *constructor* method.

Our `add` method in the `Rational` class will be used like this:

```
>>> r1.add(r2)
```

This should return a `Rational` number that is the result of adding `r1` and `r2`. So, we should be able to write:

```
>>> r3 = r1.add(r2)
```

Now, `r3` will refer to the new `Rational` number returned by the `add` method. The syntax here may struck you as funny at first, but humor us; we'll see in a moment why this syntax is sensible.

Let's write this `add` method! If `r1` $= \frac{a}{b}$ and `r2` $= \frac{c}{d}$ then `r1+r2` $= \frac{ad+bc}{bd}$. Note that the resulting fraction might be simplified by dividing out by terms that are common to the numerator and the denominator, but let's not worry about that for now. Here is the `Rational` class with its shiny new `add` method:

```
class Rational:
    def __init__(self, num, denom):
        self.numerator = num
        self.denominator = denom

    def add(self, other):
        newNumerator = self.numerator * other.denominator +
                       self.denominator * other.numerator
        newDenominator = self.denominator*other.denominator
        return Rational(newNumerator, newDenominator)
```

What's going on here!? Notice that the `add` method takes in two arguments, `self` and

`other`, while our examples above showed this method taking in a single argument. (Stop here and think about this. This is analogous to what we saw earlier with the `__init__` method.)

To sort this all out, let's consider the following sequence:

```
>>> r1 = Rational(1, 2)
>>> r2 = Rational(1, 3)
>>> r3 = r1.add(r2)
```

The instruction `r1.add(r2)` does something funky: It calls the `Rational` class's `add` method. It seems to pass in just `r2` to the `add` method but that's an optical illusion! In fact, it passes in two values: *first* it *automatically* passes a reference to `r1` and *then* it passes in `r2`. This is great, because our code for the `add` method is expecting two arguments: `self` and `other`. So, `r1` goes into the `self` "slot" and `r2` goes into the `other` slot. Now, the `add` method can operate on those two `Rational` numbers, add them, construct a new `Rational` number representing their sum, and then return that new object.

"Snazzy" is a technical term.

Here's the key: Consider some arbitrary class `Blah`. If we have an object `myBlah` of type `Blah`, then `myBlah` can invoke a method `foo` with the notation `myBlah.foo(arg1, arg2, ..., argN)`. The method `foo` will receive *first* a reference to the object `myBlah` followed by all of the `N` arguments that are passed in explicitly. Python just knows that the first argument is always the automatically-passed-in reference to the object before the "dot". The beauty of this seemingly weird system is that the method is invoked by an object and the method "knows" which object invoked it. Snazzy!

After performing the above sequence of instructions, we could type:

```
>>> r3.numerator
>>> r3.denominator
```

What we would we see? We'd see the numerator and denominator of the `Rational` number `r3`. In this case, the numerator would be 5 and the denominator would be 6.

Notice that instead of typing `r3 = r1.add(r2)` above, we could have instead have typed `r3 = r2.add(r1)`. What would have happened here? Now, `r2` would have called the `add`

method, passing `r2` in for `self` and `r1` in for `other`. We would have gotten the same result as before because addition of rationals is commutative.

6.4 Overloading

So far, we have built a basic class for representing rational numbers. It's neat and useful, but now we're about to make it even spiffier.

"Spiffy" is yet another technical term.

You probably noticed that the syntax for adding two `Rational` numbers is a bit awkward. When we add two integers, like 42 and 47, we certainly don't type `42.add(47)`, we type `42+47` instead.

It turns out that we can use the operator "+" to add `Rational` numbers too! Here's how: We simply change the name of our `add` method to `__add__`. Those are two underscore characters before and two underscore characters after the word `add`. Python has a feature that says "if a function is named `__add__` then when the user types `r1 + r2`, I will translate that into `r1.__add__(r2)`." How does Python know that the addition here is addition of `Rational` numbers rather than addition of integers (which is built-in)? It simply sees that `r1` is a `Rational`, so the "+" symbol must represent the `__add__` method in the `Rational` class. We could similarly define `__add__` methods for other classes and Python will figure out which one applies based on the type of data in front of the "+" symbol.

This feature is called *overloading*.

This is "good" overloading. "Bad" overloading involves taking more than 18 credits in a term.

We have overloaded the "+" symbol to give it a meaning that depends on the context in which it is used. Many, though not all, object-oriented programming languages support overloading. In Python, overloading addition is just the tip of the iceberg. Python allows us to overload all of the normal arithmetic operators and all of the comparison operators such as "==", "`!=`", "<", among others.

Let's think for a moment about comparing rational numbers for equality. Consider the following scenario, in which we have two different `Rational` objects and we compare them for equality:

```
>>> r1 = Rational(1, 2)
>>> r2 = Rational(1, 2)
```

```
>>> r1 == r2
False
```

Why did Python say "`False`"? The reason is that even though `r1` and `r2` *look* the same to us, each one is a reference to a *different* object. The two objects have identical contents, but they are different nonetheless, just as two identical twins are two different people. Another way of seeing this is that `r1` and `r2` refer to different blobs of memory, and when Python sees them ask if `r1 == r2` it says "Nope! Those two references are not to the same memory location." Since we haven't told Python how to compare `Rationals` in any other way, it simply compares `r1` and `r2` to see if they are referring to the very same object.

"Blob" really is *a technical term!*

So let's "overload" the "==" symbol to correspond to a function that will do the comparison as we intend. We'd like for two rational numbers to be considered equal if their ratios are the same, even if their numerators and denominators are not the same. For example $\frac{1}{2} = \frac{42}{84}$. One way to test for equality is to use the "cross-multiplying" method that we learned in grade school: Multiply the numerator of one of the fractions by the denominator of the other and check if this is equal to the other numerator-denominator product. Let's first write a method called `__eq__` to include in our `Rational` number class to test for equality.

```
def __eq__(self, other):
    return self.numerator * other.denominator ==
           self.denominator * other.numerator
```

Now, if we have two rational numbers such as `r1` and `r2`, we could invoke this method with `r1.__eq__(r2)` or with `r2.__eq__(r1)`. But because we've used the special name `__eq__` for this method, Python will know that when we write `r1 == r2` it should be translated into `r1.__eq__(r2)`. There are many other symbols that can be overloaded. (To see a complete list of the methods that Python is happy to have you overload, go to http://docs.python.org/2/reference/datamodel.html#special-method-names.)

For example, we can overload the ">=" symbol by defining a method called `__ge__` (which stands for **g**reater than or **e**qual). Just like `__eq__`, this method takes two arguments: A reference to the *calling object* that is passed in automatically (`self`) and a reference to another object to which we are making the

This example serves to doubly underscore

comparison. So, we could write our `__ge__` method as follows:

the beauty of overloading!

```
def __ge__(self, other):
    return self.numerator * other.denominator >=
           self.denominator * other.numerator
```

Notice that there is only a tiny difference between how we implemented our `__eq__` and `__ge__` methods. Take a moment to make sure you understand why `__ge__` works.

Finally, let's revisit the original fuel problem with which we started the chapter. Recall that due to numerical imprecision with floating-point numbers, we had experienced mission failure:

We hope that you aren't feeling overloaded at this point. We'd feel bad if you "object"ed to what we've done here.

```
>>> fuelNeeded = 42.0/1000
>>> tank1 = 36.0/1000
>>> tank2 = 6.0/1000
>>> tank1 + tank2 >= fuelNeeded
False
```

In contrast, we can now use our slick new `Rational` class to save the mission!

```
>>>  fuelNeeded = Rational(42, 1000)
>>>  tank1 = Rational(36, 1000)
>>>  tank2 = Rational(6, 1000)
>>>  tank1 + tank2 >= fuelNeeded
True
```

Mission accomplished!

6.5 Printing an Object

Our `Rational` class is quite useful now. But check this out:

```
>>> r1 = Rational(1, 2)
>>> r2 = Rational(1, 3)
>>> r3 = r1 + r2
>>> r3
<Rational.Rational instance at 0x6b918>
>>> print(r3)
<Rational.Rational instance at 0x6b918>
```

0x6b918!? What the heck is that?!

Notice the strange output when we asked for `r3` or when we tried to `print(r3)`. In both cases, Python is telling us, "`r3` is a `Rational` object and I've given it a special internal name called `0x` blah, blah, blah."

What we'd really like, at least when we `print(r3)`, is for Python to display the number in some nice way so that we can see it! You may recall that Python has a way to "convert" integers and floating-point numbers into strings using the built-in function `str`. For example:

```
>>> str(1)
'1'
>>> str(3.142)
'3.142'
```

So, since the `print` function wants to print strings, we can print numbers this way:

```
>>> print(str(1))
1
>>> print("My favorite number is " + str(42))
My favorite number is 42
```

In fact, other Python types such as lists and dictionaries also have `str` functions:

```
>>> myList = [1, 2, 3]
>>> print("Here is a very nice list: " + str(myList))
Here is a very nice list: [1, 2, 3]
```

Python lets us define a `str` function for our own classes by overloading a special method called `__str__`. For example, for the `Rational` class, we might write the following `__str__` method:

```
def __str__(self):
    return str(self.numerator) + "/" + str(self.denominator)
```

What is this function returning? It's a string that contains the numerator followed by a forward slash followed by the denominator. When we type `print(str(r3))`, Python will invoke this `__str__` method. That function first calls the `str` function on

`self.numerator`. Is the call `str(self.numerator)` recursive? It's not! Since `self.numerator` is an integer, Python knows to call the `str` method for integers here to get the string representation of that integer. Then, it concatenates to that string a another string containing the forward slash, `/`, indicating the fraction line. Finally, to that string it concatenates the string representation of the denominator. Now, this string is returned. So, in our running example from above where `r3` is the rational number $\frac{5}{6}$, we could use our `str` method as follows:

```
>>> r3
<Rational.Rational instance at 0x6b918>
>>> print("Here is r3: "+str(r3))
Here is r3: 5/6
```

Notice that in the first line when we ask for `r3`, Python just tells us that it is a reference to `Rational` object. In the third line, we ask to convert `r3` into a string for use in the `print` function. By the way, the `__str__` method has a closely related method named `__repr__` that you can read about on the Web.

6.6 A Few More Words on the Subject of Objects

Let's say that we wanted (for some reason) to change the numerator of `r1` from its current value to 42. We could simply type

```
r1.numerator = 42
```

In other words, the internals of a `Rational` object can be changed. Said another way, the `Rational` class is mutable. (Recall our discussion of mutability in Chapter 5.) *In Python, classes that we define ourselves are mutable (unless we add fancy special features to make them immutable).* To fully appreciate the significance of mutability of objects, consider the following pair of functions:

```
def foo():
    r = Rational(1, 3)
    bar(r)
    print r

def bar(number):
    number.numerator += 1
```

What happens when we invoke function `foo`? Notice that function `bar` is not returning anything. However, the variable `number` that it receives is presumably a `Rational` number, and `foo` increments the value of this variable's `numerator`. Since user-defined classes such as `Rational` are mutable, this means that the `Rational` object that was passed in will have its numerator changed!

How does this actually work? Notice that in the function `foo`, the variable `r` is a reference to the `Rational` number $\frac{1}{3}$. In other words, this `Rational` object is somewhere in the computer's memory and `r` is the address where this blob of memory resides. When `foo` calls `bar(r)` it is passing the reference (the memory location) `r` to `foo`. Now, the variable `number` is referring to that memory location. When Python sees `number.numerator += 1` it first goes to the memory address referred to by `number`, then uses the "dot" to look at the `numerator` part of that object, and increments that value by 1. When `bar` eventually returns control to the calling function, `foo`, the variable `r` in `foo` is still referring to that same memory location, but now the `numerator` in that memory location has the new value that `bar` set.

This brings us to a surprising fact: *Everything in Python is an object!* For example, Python's list datatype is an object. "Wait a second!" we hear you exclaim. "The syntax for using lists doesn't look anything like the syntax that we used for using `Rationals`!" You have a good point, but let's take a closer look.

Our legal team objected to us using the word "everything" here, but it's close enough to the truth that we'll go with it.

For the case of `Rationals`, we had to make a new object this way:

```
r = Rational(1, 3)
```

On the other hand, we can make a new list more simply:

```
myList = [42, 1, 3]
```

In fact, though, this list notation that you've grown to know and love is just a convenience that the designers of Python have provided for us. It's actually a shorthand for this:

```
myList = list()
myList.append(42)
myList.append(1)
```

```
myList.append(3)
```

Now, if we ask Python to show us `myList` it will show us that it is the list `[42, 1, 3]`. Notice that the line `myList = list()` is analogous to `r = Rational(1, 3)` except that we do not provide any initial values for the list. Then, the `append` method of the list class is used to append items onto the end of our list. Lists are mutable, so each of these `append`s changes the list!

Indeed, the list class has many other methods that you can learn about online. For example, the `reverse` method reverses a list. Here's an example, based on the `myList` list object that we created above:

```
>>> myList
[42, 1, 3]
>>> myList.reverse()
>>> myList
[3, 1, 42]
```

Notice that this `reverse` method is not returning a new list but rather mutating the list on which it is being invoked.

Before moving on, let's reflect for a moment on the notation that we've seen for combining two lists:

```
>>> [42, 1, 3] + [4, 5]
[42, 1, 3, 4, 5]
```

How do you think that the "+" symbol works here? You got it–it's an overloaded method in the list class! That is, it's the `__add__` method in that class!

Strings, dictionaries, and even integers and floats are all objects in Python! However, a few of these built-in types - such as strings, integers, and floats - were designed to be immutable. Recall from the previous chapter that this means that their internals cannot be changed. You can define your own objects to be immutable as well, but it requires some effort and it's rarely necessary, so we won't go there.

6.7 Getting Graphical with OOPs

We've seen that object-oriented programming is elegant and, hopefully, you now believe that it's useful. But what about 3D graphics and our video game? That's where we're headed next!

Warning! This section contains graphical language!

To get started, you'll need to get the "VPython" 3D graphics system for Python. You can find it at vpython.org.

Once you have VPython installed, import the VPython graphics module, called `visual`, as follows:

```
>>> from visual import *
```

Next, type:

```
>>> b = box()
```

What you'll see now on the screen is a white box. It might look more like a white square, so rotate it around to see that it's actually a 3D object.

As you may have surmised, `box` is a class defined in the `visual` module. The command

```
>>> b = box()
```

invoked the constructor to create a new `box` object and we made `b` be the name, or more precisely "a reference", to that box.

Just like our `Rational` number class had `numerator` and `denominator` attributes, the `box` class has a number of attributes as well. Among these are the box's length, height, and width; its position, its color, and even its material properties. Try changing these attributes at the command line as follows:

```
>>> b.length = 0.5  # the box's length just changed
>>> b.width = 2.0 # the box's width just changed
>>> b.height = 1.5  # the box's height just changed
>>> b.color = (1.0, 0.0, 0.0)  # the box turned red
```

In the display window (where the box is displayed), click and drag with the right mouse button (hold down the command key on a Macintosh). Drag left or right, and you rotate around the scene. To rotate around a horizontal axis, drag up or down. Click and drag up or down with the middle mouse button to move closer to the scene or farther away (on a 2-button mouse, hold down the left and right

```
>>> b.material = material.wood  # it's wood-grained!
>>> b.material = material.glass # it's translucent!
```

buttons; on a 1-button mouse, hold down the Option key).

When we initially made our `b = box()` it had "default" values for all of these attributes. The `length`, `width`, and `height` attributes were all 1.0, the `color` attribute was white, and the `material` attribute was a boring basic one. Notice that some of the attributes of a `box` are fairly obvious: `length`, `width`, and `height` are all numbers. However, the `color` attribute is weird. Similarly, the `material` property was set in a strange way. Let's take a closer look at just one of these attributes and you can read more about others later on the VPython documentation website.

Let's make a new box and ask Python for it's color attribute:

```
>>> c = box()
>>> c.color
(1.0, 1.0, 1.0)
```

VPython represents color using a tuple with three values, each between 0.0 and 1.0. The three elements in this tuple indicate how much red (from 0.0, which is none, to 1.0, which is maximum), green, and blue, respectively, is in the color of the object.

So, $(1.0, 1.0, 1.0)$ means that we are at maximum of each color, which amounts to bright white. The tuple $(1.0, 0.0, 0.0)$ is bright red and the tuple $(0.7, 0.0, 0.4)$ is a mixture of quite a bit of red and somewhat less blue.

If your roommate sees you staring at your fingers, just explain that you are doing something very technical.

The `box` class has another attribute called `pos` that stores the position of the center of the box. The coordinate system used by VPython is what's called a "right-handed" coordinate system: If you take your right hand and stick out your thumb, index finger, and middle finger so that they are perpendicular to one another with your palm facing you, the positive x axis is your thumb, the positive y axis is your index finger, and the positive z axis is your middle finger.

Said another way, before you start rotating in the display window with your mouse, the horizontal axis is the x axis, the vertical axis is the y axis, and the z axis points out of the screen.

When you used your mouse to rotate the scene, that actually rotated the entire coordinate system.

Take a look at the position of the box by typing `b.pos`. You'll see

this:

```
>>> b.pos
vector(0, 0, 0)
```

VPython has a class called `vector`, and `pos` is an object of this type. Nice! The `box` class is defined using the `vector` class. Using a `vector` class inside the `box` class is, well, very classy! "OK," we hear you concede grudgingly, "but what's the point of a `vector`? Why couldn't we just use a tuple or a list instead?" Here's the thing: The `vector` class has some methods defined in it for performing vector operations. For example, the `vector` class has an overloaded addition operator for adding two vectors:

Is vector class also called linear algebra?

```
>>> v = vector(1, 2, 3)
>>> w = vector(10, 20, 30)
>>> v + w
vector(11, 22, 33)
```

This class has many other vector operations as well. For example, the `norm()` method returns a `vector` that points in the same direction but has magnitude (length) 1:

```
>>> u = vector(1, 1, 0)
>>> u.norm()
vector(0.707106781186547, 0.707106781186547, 0)
```

So, while we *could* have represented vectors using lists, we wouldn't have a nice way of adding them, `norm`alizing them, and doing all kinds of other things that vectors like to do.

But, our objective for now is to change our box's `pos` vector in order to move it. We can do this, for example, as follows:

Take a look at the rich set of other vector operations on the VPython web site in order to dot your i's and cross your t's or, more precisely, to dot your scalars and cross your vectors!

```
>>> b.pos = vector(0, 1, 2)
```

While we can always create a box and change its attributes afterwards, sometimes it's convenient to just set the attributes of

the box at the time that the box is first instantiated. The `box` class constructor allows us to set the values of attributes at the time of construction like this:

```
>>> b = box(length = 0.5, width = 2.0, height = 1.5,
... color = (1.0, 0.0, 0.0))
```

Whatever attributes we don't specify will get their default values. For example, since we didn't specify a vector value for `pos`, the box's initial position will be the origin.

VPython has lots of other shape classes beyond boxes including spheres, cones, cylinders, among others. While these objects have their own particular attributes (for example a sphere has a radius), all VPython objects share some useful methods. One of these methods is called `rotate`. Not surprisingly, this method rotates its object. Let's take `rotate` out for a spin!

Try this with the box `b` that we defined above:

```
>>> b.rotate(angle=pi/4)
```

We are asking VPython to rotate the box `b` by $\frac{\pi}{4}$ radians. The rotation is, by default, specified in radians about the x axis.

Now, let's put this all together to write a few short VPython programs just to flex our 3D graphics muscles. First, let's write a *very* short program that rotates a red box forever:

```
from visual import *

def spinBox():
    myBox = box(color = (1.0, 0.0, 0.0))
    while True:
        # Slow down the animation to 60 frames per second.
        # Change the value to see the effect!
        rate(60)
        myBox.rotate(angle=pi/100)
```

Second, take a look at the program below. Try to figure out what it's doing before you run it.

```
from visual import *
import random
```

```
def spinBoxes():
    boxList = []
    for boxNumber in range(10):
        x = random.randint(-5, 5) # integer between -5,5
        y = random.randint(-5, 5)
        z = random.randint(-5, 5)
        red = random.random()     # real number between 0 & 1
        green = random.random()
        blue = random.random()
        newBox = box(pos = vector(x, y, z),
                     color = (red, green, blue) )
        boxList.append(newBox)
    while True:
        for myBox in boxList:
            rate(60)
            myBox.rotate(angle=pi/100)
```

This is very cool! We now have a list of objects and we can go through that list and rotate each of them.

6.8 Robot and Zombies, Finally!

It's time to make our video game! The premise of our game is that we will control a robot that moves on the surface of a disk (a large flat cylinder) populated by zombies. The player will control the direction of the robot with the keyboard: Pressing the "j" key will turn the robot a bit to the left and pressing the "k" key will turn it a bit to the right. The zombies will each turn independently by random amounts.

The program will be in an infinite loop. At each step, the player's robot will check to see if the player has pressed a key to make it turn. Then, whether or not it turns, the robot will take a small step forward. Similarly, each zombie will turn a random amount and then take a small step forward. Our game will have no particular objective, but you can add one later if you like. Perhaps, the objective is to run into as many zombies as possible – or perhaps avoid them.

To get started, we wish to define a player robot class that we'll use to manufacture ("instantiate") the player's robot and another class that allows to instantiate zombies. It particularly makes sense to have a zombie *class* because a class allows us to instantiate many *objects* – and we indeed plan to have many zombies!

In fact, the player's robot and zombies have a lot in common. They are 3D entities that should be able to move forward and turn. Because of this commonality, we would be

replicating a lot of effort if we were to define a robot class and a zombie class entirely separately. On the other hand, the two classes are not going to be identical because the player's robot looks different from zombies (we hope) and because the robot will be controlled by the player while the zombies move on their own.

So, here's the key idea: We'll define a class – let's call it `GenericBot` – that has all of the attributes that any entity in our game – a player robot or a zombie – should have. Then, we'll define a `PlayerBot` class and a `ZombieBot` class both of which "inherit" all of the attributes and methods of the `GenericBot` and *add* the special extras (e.g., how their bodies look) that differentiate them.

Our `GenericBot` class will have a constructor, an `__init__` method, that takes as input the initial position of the bot, its initial heading (the direction in which it is pointing), and its speed (the size of each step that it makes when we ask it to move forward). Here's the code; we'll dissect it below.

```
from visual import *
import math
import random

class GenericBot:
    def __init__(self, position = vector(0, 0, 0),
                 heading = vector(0, 0, 1), speed = 1):
        self.location = position
        self.heading = heading.norm()
        self.speed = speed
        self.parts = []

    def update(self):
         self.turn(0)
         self.forward(self.speed)

    def turn(self, angle):
        # convert angle from degrees to radians (VPython
        # assumes all angles are in radians)
        theta = math.radians(angle)
        self.heading = rotate(self.heading, angle = theta,
                              axis = vector(0, 1, 0))
        for part in self.parts:
            part.rotate(angle = theta, axis = vector(0, 1, 0),
                        origin = self.position)

    def forward(self):
        self.position += self.heading * self.speed
        for part in self.parts:
            part.pos += self.heading * self.speed
```

Let's start with the `__init__` method – the so-called "constructor" method. It takes three input arguments: A `position` (a VPython `vector` object indicating the bot's initial position), a `heading` (a vector indicating the direction that the bot is initially pointing), and `speed` (a number indicating how far the bot moves at each update step). Notice that the line:

```
def __init__(self, position = vector(0, 0, 0),
             heading = vector(0, 0, 1), speed = 1):
```

provides *default values* for these inputs. This means that if the user doesn't provide values for these input arguments, the inputs will be set to these values. (You may recall that the `box` class had default arguments as well. We could either define a new box with `b = box()` in which case we got the default values or we could specify our own values for these arguments.) If the user provides only some of the input arguments, Python will assume that they are the arguments from left-to-right. For example, if we type

```
>>> mybot = GenericBot(vector(1, 2, 3))
```

then Python assumes that `vector(1, 2, 3)` should go into the `position` variable and it uses the default values for `heading` and `speed`. If we type

```
>>> mybot = GenericBot(vector(1, 2, 3), vector(0, 0, 1))
```

then the first vector goes into the `position` argument and the second goes into the `heading` argument. If we want to provide values in violation of the left-to-right order, we can always tell Python which value we are referring to like this:

```
>>> mybot = GenericBot(heading=vector(0, 1, 0))
```

Now, Python sets the `heading` to the given value and uses the default values for the other arguments.

It was de-fault of de-authors for this de-gression!

OK, so much for defaults! The `__init__` method then sets its location (`self.location`), heading (`self.heading`), speed (`self.speed`), and parts (`self.parts`) attributes. The `self.heading` is normalized to make the length of the vector a

unit vector (a vector of length 1) using the `vector` class's `norm()` method. The `self.parts` list will be a list of VPython 3D objects – boxes, spheres, and so forth – that make up the body of the bot. Since the player bot and the zombie bots will look different, we haven't placed any of these body parts into the list just yet. That's coming soon!

Notice that the `GenericBot` has three additional methods: `update`, `forward`, and `turn`. In fact, the `update` method simply calls the `turn` method to turn the bot 0 radians (we'll probably change that later!) and then calls the `forward` method to move one step at the given speed. The `turn` method changes the bot's `self.heading` so that it heads in the new direction induced by the turning angle and then rotates each of the parts in the bot's `self.parts` list by that same angle.

Something **very lovely** and subtle is happening in the `for` loop of the `turn` method! Notice that each `part` is expected to be a VPython object, like a `box` or `sphere` or something else. Each of those objects has a `rotate` method. Python is saying here "hey `part`, figure out what kind of object you are and then call your `rotate` method to rotate yourself." So, if the first `part` is a `box`, then this will call the `box` `rotate` method. If the next part is a `sphere`, the `sphere`'s `rotate` method will be called here. This all works great, as long as each element in the `self.parts` list has a `rotate` method. Fortunately, all VPython shapes do have a `rotate` method.

We'll also just point out quickly that the line

```
part.rotate(angle = theta, axis = vector(0, 1, 0),
            origin = self.position)
```

is telling that `part` to rotate by an angle `theta` about a vector aligned with `vector (0, 1, 0)` (the y-axis) but starting at the vector given by `self.position`. Under our assumption that the y-axis is "up", this effectively rotates the object about a line that runs through the center of its body from "its feet to its head". That is, it rotates the body parts the way we would like it to rotate, as opposed to the default rotation which is about the x-axis.

The `forward` method changes the bot's `self.position` vector by adding to it the `heading` vector scaled by the bot's `self.speed`. Note that `self.position` is just the bot's own self-concept of where it is located. We also need to physically move all of the parts of the bot's body, which is done by changing the `pos` position vector of each VPython objects in the `self.parts` list, again by the `self.heading` vector scaled by `self.speed`.

Next comes the most amazing part of this whole business! We now define the `ZombieBot` class that *inherits* all of the methods and attributes of the `GenericBot` but adds the components that are specific to a zombie. Here's the code and we'll discuss it in a moment.

Drumroll, please!

```
class ZombieBot(GenericBot):
    def __init__(self, position = vector(0, 0, 0),
                 heading = vector(0, 0, 1)):
        GenericBot.__init__(self, position, heading)
        self.body = cylinder(pos = self.location,
                             axis = (0, 4, 0), radius = 1,
                             color = (0, 1, 0))
        self.arm1 = cylinder(pos = self.location +
                                        vector(0.6, 3, 0),
                             axis = (0, 0, 2), radius = .3,
                             color = (1, 1, 0))
        self.arm2 = cylinder(pos = self.location +
                                        vector(-0.6, 3, 0),
                             axis = (0, 0, 2), radius = .3,
                             color = (1, 1, 0))
        self.halo = ring(pos = self.location +
                                        vector(0, 5, 0),
                             axis = (0, 1, 0), radius = 1,
                             color = (1, 1, 0))
        self.head = sphere(pos = self.location +
                                        vector(0, 4.5, 0),
                             radius = 0.5, color = (1, 1, 1))
        self.parts = [self.body, self.arm1, self.arm2,
                      self.halo, self.head]

    def update(self):
        # call turn with a random angle between -5 and 5
        # degrees
        self.turn(random.uniform(-5, 5))
        self.forward()
```

The class definition begins with the line: `class ZombieBot(GenericBot)`. The `GenericBot` in parentheses is saying to Python "this class inherits from `GenericBot`." Said another way, a `ZombieBot` "is a kind of" `GenericBot`. Specifically, this means that a `ZombieBot` has the `__init__`, `update`, `turn`, and `forward` methods of `GenericBot`. The class `GenericBot` is called the *superclass* of `ZombieBot`. Similarly, `ZombieBot` is called a *subclass* or *derived class* of `GenericBot`.

Note that `ZombieBot` has its own `__init__` constructor method. If we had not defined this `__init__` then each time we constructed a `ZombieBot` object, Python would

automatically invoke the `__init__` from `GenericBot`, the superclass from which `ZombieBot` was derived. However, since we've defined an `__init__` method for `ZombieBot`, that method will get called when we instantiate a `ZombieBot` object. It's not that the `GenericBot`'s constructor isn't useful to us, but we want to do some other things too. Specifically, we want to populate the list of body parts, `self.parts`, with the VPython shapes that constitute a zombie.

We get two-for-the-price-of-one by first having the `ZombieBot`'s `__init__` method call the `GenericBot`'s `__init__` method to do what it can do for us. This is invoked via `GenericBot.__init__(self, position, heading)`. This is saying, "hey, I know that I'm a `ZombieBot`, but that means I'm a kind of `GenericBot` and, as such, I can call any `GenericBot` method for help. In particular, since the `GenericBot` already has an `__init__` method that does some useful things, I'll call it to set my `position` and `heading` attributes."

After calling the `GenericBot` constructor, the `ZombieBot` constructor continues to do some things on its own. In particular, it defines some VPython objects and places them in the `parts` list of body parts. You might notice that all of those body parts are positioned relative to the `ZombieBot`'s `position` – which is just a vector that we defined that keeps track of where the bot is located in space.

Since `ZombieBot` inherited from `GenericBot`, it automatically has the `update`, `turn`, and `forward` methods defined in the `GenericBot` class. The `turn` and `forward` methods are fine, but the `update` method needs to be replaced to turn the `ZombieBot` at random. Thus, we provide a new `turn` method in the `ZombieBot` class.

Now, imagine that we do the following:

```
>>> zephyr = ZombieBot()
>>> zephyr.update()
```

The first line creates a new `ZombieBot` object. Since we didn't provide any inputs to the constructor, the default values are used and zephyr the zombie is at position (0, 0, 0) and heading in direction (0, 0, 1). The second line tells zephyr to update itself. Python checks to see if the `ZombieBot` class contains an `update` method. It does, so that method is invoked. That method then calls the `turn` method with a random angle between -5 and 5 degrees. Python checks to see if the `ZombieBot` class has its own `turn` method. Since it doesn't, Python goes to the superclass, `GenericBot`, and looks for a `turn` method there.

There is one there and that's what's used! Next, the `update` method calls the `forward` method. Since there's no `forward` method defined in `ZombieBot`, Python again goes to the superclass and uses the `forward` method there.

The very nice thing here is that `ZombieBot` inherits many things from the superclass from which it is derived and only changes – or *overrides* – those methods that it needs to customize for zombies.

Now, we can do something similar for the player's robot:

```
class PlayerBot(GenericBot):
    def __init__(self, position = vector(0, 0, 0),
                 heading = vector(0, 0, 1)):
        GenericBot.__init__(self, position, heading)
        self.body = cylinder(pos = self.location +
                                        vector(0, 0.5, 0),
                             axis = (0, 6, 0), radius = 1,
                             color = (1, 0, 0))
        self.head = box(pos = vector(0, 7, 0) + self.location,
                        length = 2, width = 2, height = 2,
                        color = (0, 1, 0))
        self.nose = cone(pos = vector(0, 7, 1) +
                                 self.location,
                         radius = 0.5, axis = (0, 0, 1),
                         color = (1, 1, 0))
        self.wheel1 = cylinder(pos = self.location +
                                         vector(1, 1, 0),
                               axis = (0.5, 0, 0), radius = 1,
                               color = (0, 0, 1))
        self.wheel2 = cylinder(pos = self.location +
                                         vector(-1, 1, 0),
                               axis = (-0.5, 0, 0),
                               radius = 1, color = (0, 0, 1))
        self.parts = [self.body, self.head, self.nose,
                      self.wheel1, self.wheel2]

    def update(self):
        if scene.kb.keys: # is there an event waiting to be
                          # processed?
            s = scene.kb.getkey() # get keyboard information
            if s == "l":
                self.turn(5)
            elif s == "k":
                self.turn(-5)
        self.forward()
```

The `PlayerBot` class also inherits from the `GenericBot` class. It again calls the `GenericBot`'s constructor for help initializing some attributes and then defines its own

attributes. It uses the `turn` and `forward` methods from the superclass, but it changes the `update` method to get input from the user. We won't dissect the details of how VPython deals with keyboard input, but you can use this code and modify or augment it to handle other keyboard inputs from the player.

Recall that a short awhile ago we noted that all Python shapes have a `rotate` method. That's because all of these shapes – boxes, spheres, cylinders, cones, and others – inherit from a shape superclass that defines a `rotate` method. Therefore, they all "know" how to rotate because they inherited that "knowledge" from their parent class. That parent class has many features – methods and attributes – that its children need, just like our `GenericBot` class has features that are used by its children – the derived classes `ZombieBot` and `PlayerBot`.

Finally, here's our game! In this file, we import the VPython (`visual`) graphics package; the `robot.py` file containing the `GenericBot`, `ZombieBot`, and `PlayerBot` classes, and the `random` and `math` packages. The `main` function instantiates a large flat VPython cylinder object, that we name `ground`, that is the surface on which the bots will move. It's radius is given by a global variable `GROUND_RADIUS`. We then instantiate a single `PlayerBot` named `player` and call a helper function called `makeZombies` that instantiates many zombies (the number is given by the global variable `ZOMBIES`) and returns a list of `ZombieBot` objects at random positions on our cylindrical playing area. Finally, the `main` function enters an infinite loop. At each iteration, it calls the `PlayerBot`'s `update` method to move. This method checks to see if the user has pressed the keyboard to turn the robot and also moves forward one step. We check to see if the player's location is beyond the radius of the playing area, and if so turn the robot 180 degrees so that its next step will hopefully be inside the player area. Now, each zombie is updated by calling its `update` method. Here too, if the zombie has stepped out of the playing area, we rotate it at random (180 degrees plus or minus 30 degrees). Voilá!

```
from visual import *
from robot import *
import random
import math

GROUND_RADIUS = 50
ZOMBIES = 20

def main():
    ground = cylinder(pos = (0, -1, 0), axis = (0, 1, 0),
                      radius = GROUND_RADIUS)
    player = PlayerBot()
```

```
    zombies = makeZombies()
    while True:
        rate(30)
        player.update()
        if mag(player.location) >= GROUND_RADIUS:
            player.turn(180)
        for z in zombies:
            z.update()
            if mag(z.location) >= GROUND_RADIUS:
                z.turn(random.uniform(150, 210))

def makeZombies():
    zombies = []
    for z in range(ZOMBIES):
        theta = random.uniform(0, 360)
        r = random.uniform(0, GROUND_RADIUS)
        x = r * cos(math.radians(theta))
        z = r * sin(math.radians(theta))
        zombies.append(ZombieBot(position = vector(x, 0, z)))
    return zombies
```

6.9 Conclusion

I suppose that "yucky" is yet another technical term.

This is all really neat, but why is object-oriented programming such a big deal? As we saw in our `Rational` example, one benefit of object-oriented programming is that it allows us to define new types of data. You might argue, "Sure, but I could have represented a rational number as a list or tuple of two items and then I could have written functions for doing comparisons, addition, and so forth without using any of this class stuff." You're absolutely right, but you then have exposed a lot of yucky details to the user that she or he doesn't want to know about. For example, the user would need to know that rational numbers are represented as a list or a tuple and would need to remember the conventions for using your comparison and addition functions. One of the beautiful things about object-oriented programming is that all of this "yuckiness" (more technically, "implementation details") is *hidden* from the user, providing a *layer of abstraction* between the use and the implementation of rational numbers.

Layer of abstraction?! What does *that* mean? Imagine that every time you sat in the driver's seat of a car you had to fully understand various components of the engine, transmission, steering system, and electronics just to operate the car. Fortunately, the designers of cars have presented us with a nice layer of abstraction: the steering wheel, pedals, and dashboard. We can now do interesting things with our car without having to

think about the low-level details. As a driver, we don't need to worry about whether the steering system uses a rack and pinion or something entirely different. This is precisely what classes provide for us. The inner workings of a class are securely "under the hood," available if needed, but not the center of attention. The user of the class doesn't need to worry about implementation details; she or he just uses the convenient and intuitive provided methods. By the way, the "user" of your class is most often you! You too don't want to be bothered with implementation details when you use the class—you'd rather be thinking about bigger and better things at that point in your programming.

Object-oriented design is the computer science version of *modular design*, an idea that engineers pioneered long ago and have used with great success. Classes are modules. They encapsulate logical functionality and allow us to reason about and use that functionality without having to keep track of every part of the program at all times. Moreover, once we have designed a good module/class we can reuse it in many different applications.

Finally, in our Robot and Zombies game, we saw the important idea of inheritance. Once we construct one class, we can write special versions that inherit all of the methods and attributes of the "parent" or superclass, but also add their own unique features. In large software systems, there can be a large and deep *hierarchy* of classes: One class has children classes that inherit from it which in turn have their own children classes, and so forth. This design methodology allows for great efficiencies in reusing rather than rewriting code.

Takeaway message: *Classes—the building blocks of object-oriented designs and programs—provide us with a way of providing abstraction that allows us to concentrate on using these building blocks without having to worry about the internal details of how they work. Moreover, once we have a good building block we can use it over and over in all different kinds of programs.*

Chapter 7: How Hard is the Problem?

> *It is a mistake to think you can solve any major problems just with potatoes.*
>
> —Douglas Adams

7.1 The Never-ending Program

Chances are you've written a program that didn't work as intended. Many of us have had the particularly frustrating experience of running our new program and observing that it seems to be running for a very long time without producing the output that we expected. "Hmmm, it's been running now for nearly a minute. Should I stop the program? Or, perhaps, since I've invested this much time already I should give the program another minute to see if it's going to finish," you say to yourself. Another minute passes and you ask yourself if you should let it run longer or bail out now. Ultimately, you might decide that the program is probably stuck in some sort of loop and it isn't going to halt, so you press Control-C and the program stops. You wonder though if perhaps the program was just a moment away from giving you the right answer. After all, the problem might just inherently require a lot of computing time.

Wouldn't it be nice to have some way to check if your program is eventually going to halt?

Then, if you knew that your program wasn't going to halt, you could work on debugging it rather than wasting your time watching it run and run and run.

The answer to that question is "yes!" Of course that would be nice!

In fact, perhaps we could even write a program, let's call it a "halt checker," that would take *any* program as input and simply return a Boolean: `True` if the input program eventually halts and `False` otherwise. Recall from Chapter 3 that functions can take other functions as input. So there is nothing really strange about giving the hypothetical halt checker function another function as input. However, if you're not totally comfortable with that idea, another alternative is that we would give the halt checker a string as input and that string would contain the code for the Python function that we want to check. Then, the halt checker would somehow determine if the function in that string eventually halts.

For example, consider the following string which we've named `myProgram`:

```
myProgram = 'def foo(): \
        return foo()'
```

This string contains the Python code for a function called `foo`. Clearly, that function runs forever because the function calls itself recursively and there is no base case to make it stop. Therefore, if we were to run our hypothetical `haltChecker` function it should return `False`:

```
>>> haltChecker(myProgram)
False
```

How would we write such a halt checker? It would be easy to check for certain kinds of obvious problems, like the problem in the program `foo` in the example above. It seems though that it might be quite difficult to write a halt checker that could reliably evaluate *any* possible program that we would give it. After all, some programs are very complicated with all kinds of recursion, `for` loops, `while` loops, etc. Is it even possible to write such a halt checker program? In fact, in this chapter we'll show that this problem is impossible to solve on a computer. That is, it is not possible to write a halt checker program that would tell us whether *any* other program eventually halts or not.

How can we say it's impossible!? That seems like an irresponsible statement to make. After all, just because nobody has succeeded in writing such a program yet doesn't allow us to conclude that it's impossible. You've got a good point there – we agree! However, the task of writing a halt checker truly is *impossible* – it doesn't exist now and it will never exist. In this chapter we'll be able to prove that beyond a shadow of a doubt.

7.2 Three Kinds of Problems: Easy, Hard, and Impossible.

Computer scientists are interested in measuring the "hardness" of computational problems in order to understand how much time (or memory, or some other precious resource) is required for their solution. Generally speaking, time is the most precious resource so we want to know how much time it will take to solve a given problem. Roughly speaking, we can categorize problems into three groups: "easy", "hard", or "impossible."

An "easy" problem is one for which there exists a program – or algorithm – that is fast enough that we can solve the problem in a reasonable amount of time. All of the problems that we've considered so far in this book have been of that type. No program that we've written here has taken days or years of computer time to solve. That's *not* to say that coming up with the program was always easy for us, the computer scientist. That's not what we mean by "easy" in this context. Rather, we mean that there *exists* an algorithm that runs fast enough that the problem can be solved in a reasonable amount of time. You might ask, "what do you mean by *reasonable* ?" That's a reasonable question and we'll come back to that shortly.

Recall that an "algorithm" is a computational recipe. It is more general than a "program" because it isn't specific to Python or any other language. However, a program implements an algorithm.

In contrast, a "hard" problem is one for which we can find an algorithm to solve it, but every algorithm that we can find is so slow as to make the program practically useless. And the "impossible" ones, as we suggested in the introduction to this chapter, are indeed just that - absolutely, provably, impossible to solve no matter how much time we're willing to let our computers crank away on them!

7.2.1 Easy Problems

Consider the problem of taking a list of N numbers and finding the smallest number in that list. One simple algorithm just "cruises" through the list, keeping track of the smallest number seen so far. We'll end up looking at each number in the list once, and thus the number of steps required to find the smallest number is roughly N. A computer scientist would say that the running time of this algorithm "grows proportionally to N."

In Chapter 5 we developed an algorithm, called *selection sort*, for sorting items in ascending order. If you don't remember it, don't worry. Here's how it works in a nutshell: Imagine that we have N items in a list– we'll assume that they are numbers for simplicity - and they are given to us in no particular order. Our goal is to sort them from smallest to largest. In Chapter 5, we needed that sorting algorithm as one step in our music recommendation system.

Here's what the *selection sort* algorithm does. It first "cruises" through the list, looking for the smallest element. As we observed a moment ago, this takes roughly N steps. Now the algorithm knows the smallest element and it puts that element in the first position in the list by simply swapping the first element in the list with the smallest element in the list. (Of course, it might be that the first element in the list *is* the smallest element in the list, in

which case that swap effectively does nothing - but in any case we are guaranteed that the first element in the list now is the correct smallest element in the list.)

In the next phase of the algorithm, we're looking for the second smallest element in the list. In other words, we're looking for the smallest element in the list excluding the element in the first position which is now known to be the first smallest element in the list. Thus, we can start "cruising" from the second element in the list and this second phase will take $N-1$ steps. The next phase will take $N-2$ steps, then $N-3$, and all the way down to 1. So, the total number of steps taken by this algorithms is $N+(N-1)+(N-2)+\cdots+1$. There are N terms in that sum, each of which is at most N. Therefore, the total sum is certainly less than N^2. It turns out, and it's not too hard to show, the sum is actually $\frac{N(N+1)}{2}$ which is approximately $\frac{N^2}{2}$.

A computer scientist would say that the running time of this algorithm "grows proportionally to N^2." It's not that the running time is necessarily exactly N^2, but whether it's $\frac{1}{2}N^2$ or $42N^2$, the N^2 term dictates the shape of the curve that we would get if we plotted running time as a function of N.

You might hear a computer scientist say: "The running time is big-Oh of N^2" - written $O(N^2)$. That's CS-talk for what we're talking about here.

As another example, the problem of multiplying two $N \times N$ matrices using the method that you may have seen in an earlier math course takes time proportional to N^3. All three of these algorithms - finding the minimum element, sorting, and multiplying matrices – have running times of the form N^k where k is some number: We saw that $k=1$ for finding the minimum, $k=2$ for selection sorting, and $k=3$ for our matrix multiplication algorithm. Algorithms that run in time proportional to N^k are said to run in *polynomial time* because N^k is a polynomial of degree k. In practice, polynomial time is a reasonable amount of time. Although you might argue that an algorithm that runs in N^{42} steps is nothing to brag about, in fact using polynomial time as our definition of "reasonable" time is, um, er, reasonable – for reasons that will be apparent shortly.

7.2.2 Hard Problems

Imagine now that you are a salesperson who needs to travel to a set of cities to show your products to potential customers. The good news is that there is a direct flight between *every pair of cities* and, for each pair, you are given the cost of flying between those two cities. Your objective is to start in your home city, visit each city *exactly once*,

and return back home at lowest total cost. For example, consider the set of cities and flights shown in Figure 7.1 and imagine that your start city is Aville.

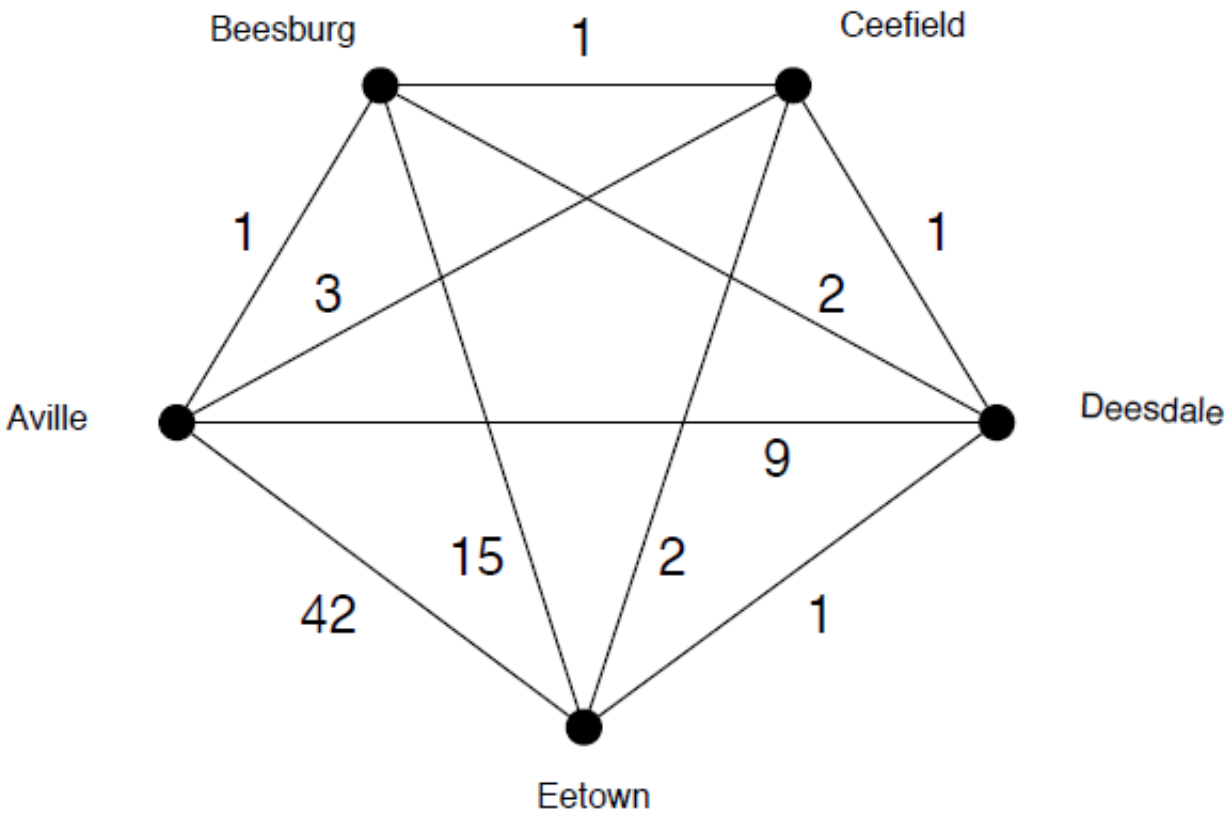


Figure 7.1: Cities and flight costs.

A tempting approach to solving this problem is to use an approach like this: Starting at our home city, Aville, fly on the cheapest flight. That's the flight of cost 1 to Beesburg. From Beesburg, we could fly on the least expensive flight to a city that we have not yet visited, in this case Ceefield. From Ceefield we would then fly on the cheapest flight to a city that we have not yet visited. (Remember, the problem stipulates that you only fly to a city once, presumably because you're busy and you don't want to fly to any city more than once - even if it might be cheaper to do so.) So now, we fly from Ceefield to Deesdale and from there to Eetown. Uh oh! Now, the constraint that we don't fly to a city twice means that we are forced to fly from Eetown to Aville at a cost of 42. The total cost of this "tour" of the cities is $1 + 1 + 1 + 1 + 42 = 46$. This approach is called a "greedy algorithm" because at each step it tries to do what looks best at the moment, without considering the long-term implications of that decision. This greedy algorithm didn't do so well here. For example, a much better solution goes from Aville to Beesburg to Deesdale to Eetown to Ceefield to Aville with a total cost of $1 + 2 + 1 + 2 + 3 = 9$. In general, greedy algorithms are fast but often fail to find optimal or even particularly good solutions.

It turns out that finding the optimal tour for the Traveling Salesperson Problem is very difficult. Of course, we could simply enumerate every one of the possible different tours, evaluate the cost of each one, and then find the one of least cost. If we were to take this approach in a situation with N cities, how many different tours would we have to explore?

Running time	Size N = 10	20	30	40
N	0.00000001	0.00000002	0.00000003	0.00000004
N^2	0.00000010	0.00000040	0.00000090	0.00000160
N^3	0.00000100	0.00000800	0.00002700	0.00006400
N^5	0.00010000	0.00320000	0.02430000	0.10240000
$N!$	0.0036	77.1 years	8400 trillion years	2.5×10^{31} years

Table 7.1: Time to solve problems of various sizes using algorithms with different running times on a computer capable of performing one billion operations per second. Times are in seconds unless indicated otherwise.

Notice that there are $N-1$ first choices for the first city to visit. From there, there are $N-2$ choices for the next city, then $N-3$ for the third, etc. Therefore there are a total of $(N-1) \times (N-2) \times (N-3) \cdots \times 1$. That product is called "$N-1$ factorial" and denoted $(N-1)!$. The exclamation point is appropriate because that quantity grows very rapidly. While $5!$ is a modest 120, $10!$ is over 3 million and $15!$ is over one trillion. Computers are fast, but examining one trillion different tours would take a long time on even the fastest computer.

We're using $N!$ rather than $(N-1)!$ here for simplicity. They only differ by a factor of N, which is negligible in the grand scheme of things.

Table 7.1 demonstrates just how bad a function like $N!$ is in comparison to polynomial functions like N, N^2, N^3, and N^5. In this table, we are imagining a computer that is capable of performing one billion "operations" per second so that our algorithm for finding the minimum element in a list in time proportional to N can find the minimum in a list of one billion items in one second. We're being pretty generous here, and assuming that this computer can also enumerate one billion traveling salesperson tours in one second, but this will only make our case against the brute-force enumeration of traveling salesperson tours even stronger!

That's a lot of exclamation marks!

Ouch! $N!$ is clearly very bad! One might be tempted to believe that in a few years, when computers get faster, this will no longer be a problem. Sadly, functions like $N!$ grow so fast that a computer that is 10 or 100 or 1000 times faster provide us with very little relief. For example, while our $N!$ algorithm takes 8400 trillion years for a problem of size 30 on our current one billion operations per second computer, it will take 8.4 trillion years on a computer that is 1000 times faster. That's hardly good news!

The Traveling Salesperson Problem is just one of many problems for which algorithms exist but are much too slow to be useful - regardless of how fast computers become in the future. In fact, the Traveling Salesperson Problem is in a group - or "class" - of problems called the *NP-hard* problems. Nobody knows how to solve any of these problems efficiently (that is, in polynomial time like N or N^2 or N^k for any constant k). Moreover, these problems all have the property that if any *one* of them can be solved efficiently (in polynomial time) then, amazingly, *all* of these problems can be solved efficiently.

Metaphorically, we can think of the NP-hard problems as a giant circle of very big dominoes - one domino for the Traveling Salesperson Person and one for every NP-hard problem. We don't know how to knock any of these dominoes over (metaphorically, how to solve any of them efficiently) *but* we know that if any one of them falls over, all of the rest of them will fall (be solvable by an efficient algorithm) as well.

Unfortunately, many important and interesting problems are NP-hard. For example, the problem of determining how a protein folds in three dimensions is NP-hard. That's truly unfortunate because if we could predict how proteins fold, we could use that information to design better drugs and combat a variety of diseases. As another example, imagine that we have a number of items of different size that we want to pack into shipping containers of some given size. How should we pack our items into shipping containers so as to minimize the number of containers used? This problem too is NP-hard. Many games and puzzles are also NP-hard. For example, the problem of determining whether large Sudoku puzzles are solvable is NP-hard as are problems related to solving Minesweeper games and many others.

The good news is that just because a problem is NP-hard does not mean that we can't find reasonably good - albeit probably not perfect - solutions. Computer scientists work on a variety of strategies for dealing with NP-hard problems. One strategy is something called *approximation algorithms*. An approximation algorithm is an algorithm that runs fast (in polynomial time) and finds solutions that are guaranteed to be within a certain percentage of the best possible solution. For example, for certain types of Traveling

Salesperson Problems we can quickly find solutions that are guaranteed to be no more than 50% more expensive than the best possible tour. You might reasonably ask, “How can you guarantee that a solution is within 50% of optimal when you have no way of efficiently finding the optimal solution?” This is indeed surprising and seemingly magical, but in fact such guarantees can be made. This topic is often studied in an “Algorithms” course.

There are many other approaches as well to dealing with NP-hard problems. One approach is called *heuristic* design. A heuristic is essentially a “rule of thumb” for approaching a problem. For example, for the Traveling Salesperson Problem, you might use the “greedy” heuristic that we mentioned earlier, namely, start by visiting the city that you can get to most cheaply. From there, visit the cheapest city that you get to that you haven’t visited yet. Continue in this fashion until you’ve visited each city once and then fly back home. That’s a simple rule and it often finds reasonably good solutions, although it sometimes does quite badly as we saw earlier. In contrast to approximation algorithms, heuristics don’t make any promises on how well they will do. There are many other approaches for dealing with NP-hard problems and this is an active area of research in computer science. For an example of an interesting biologically inspired technique for dealing with NP-hard problems, see below.

That’s a shameless sales pitch for another course if there ever was one!

Genetic Algorithms: A Biologically-Inspired Approach for Dealing with NP-hard Problems.

One technique for dealing with NP-hard problems is borrowed from biology. The idea is to “evolve” reasonably good solutions to our problem by simulating evolution on a population of possible solutions to our problem. Returning to our Traveling Salesperson example, let’s call the cities in Figure 7.1 by their first letters: A, B, C, D and E. We can represent a tour by a sequence of those letters in some order, beginning with A and with each letter appearing exactly once. For example, the tour Aville to Beesburg to Deesdale to Eetown to Ceefield and back to Aville would be represented as the sequence $ABDEC$. Notice that we don’t include the A at the end because it is implied that we will return to A at the end.

Now, let’s imagine a collection of some number of orderings such as $ABDEC$, $ADBCE$, $AECDB$, and $AEBDC$. Let’s think of each such ordering as an “organism” and the collection of these orderings as a

"population". Pursuing this biological metaphor further, we can evaluate the "fitness" of each organism/ordering by simply computing the cost of flying between the cities in that given order.

Now let's push this idea one step further. We start with a population of organisms/orderings. We evaluate the fitness of each organism/ordering. Now, some fraction of the most fit organisms "mate", resulting in new "child" orderings where each child has some attributes from each of its "parents." We now construct a new population of such children for the next generation. Hopefully, the next generation will be more fit - that is, it will, on average, have less expensive tours. We repeat this process for some number of generations, keeping track of the most fit organism (least cost tour) that we have found and report this tour at the end.

"That's a cute idea," we hear you say, "but what's all this about mating Traveling Salesperson orderings?" That's a good question - we're glad you asked! There are many possible ways we could define the process by which two parent orderings give rise to a child ordering. For the sake of example, we'll describe a very simple (and not very sophisticated) method; better methods have been proposed and used in practice.

Imagine that we select two parent orderings from our current population to reproduce (we assume that any two orderings can mate): $ABDEC$ and $ACDEB$. We choose some point at which to split the first parent's sequence in two, for example as $ABD|EC$. The offspring ordering receives ABD from this parent. The remaining two cities to visit are E and C. In order to get some of the second parent's "genome" in this offspring, we put E and C in the order in which they appear in the second parent. In our example, the second parent is $ACDEB$ and C appears before E, so the offspring is $ABDCE$.

Let's do one more example. We could have also chosen $ACDEB$ as the parent to split, and split it at $AC|DEB$, for example. Now we take the AC from this parent. In the other parent, $ABDEC$, the remaining cities DEB appear in the order BDE, so the offspring would be $ACBDE$.

In summary, a genetic algorithm is a computational technique that is effectively a simulation of evolution with natural selection. The technique allows us to find good solutions to hard computational problems by imagining candidate solutions to be metaphorical organisms and collections of such

organisms to be populations. The population will generally not include every possible "organism" because there are usually far too many! Instead, the population comprises a relatively small sample of organisms and this population evolves over time until we (hopefully!) obtain very fit organisms (that is, very good solutions) to our problem.

7.3 Impossible Problems!

So far, we've talked about "easy" problems –those that can be solved in polynomial time - and "hard" problems - those that can be solved but seemingly require impractically huge amounts of time to solve. Now we turn to problems that are downright impossible to solve, no matter how much time we are willing to spend.

First, we'd like to establish that there even exist impossible problems. We'll do this, in essence, by showing that the number of computational problems is much larger than the number of different programs. Therefore, there must be some problems for which there exist no programs. This is a strange and beautiful idea. It's strange because it ultimately will allow us to establish that there are problems that can't be solved by programs but it doesn't actually tell us what any of those problems are! This is what a mathematician calls an *existence proof*: We show that something exists (in this case uncomputable problems) without actually identifying a concrete example! The idea is also beautiful because it uses the notion of different sizes of infinities. In a nutshell, it turns out that there are an infinite number of different programs that we could write but there is a *larger* infinite number of different computational problems. "Larger infinities!?" we hear you exclaim. "Have you professors totally lost your minds?" Perhaps, but that's not relevant here. Let's dive in to our adventure in infinities, large and small.

7.3.1 "Small" Infinities

Imagine that we gave you three jelly beans. Of course, you are good at counting and you instantly recognize that you have three jelly beans, but pardon us for a moment as we look at this another way. You have three jelly beans because you can match up your jelly beans with the set of counting numbers $\{1, 2, 3\}$. A mathematician would say that there is a *bijection* – or "perfect matching" – between your set of three jelly beans and the set of

Sorry, you'll just have to imagine that. After all, if we actually gave them

counting numbers $\{1, 2, 3\}$.

to you, you'd probably eat them.

More precisely, a bijection is a matching of elements from one set (e.g., our set of jelly beans) to another set (e.g., our set of numbers $\{1, 2, 3\}$) such that every element in the *first set* is matched to a *distinct* element in the second set and every element in the *second set* is matched to something from the first set. In other words, a bijection is a "perfect matching" of elements between our two sets. We say that two sets have the same *cardinality* (or "size") if there is a bijection between them.

That seems pretty obvious and more than a bit pedantic. But here's where it gets interesting! Once we accept that two sets have the same cardinality if there exists a bijection between them, it's natural to see what happens when the two sets are infinite. Consider, for example, the set of counting numbers $\{1, 2, 3, 4, \ldots\}$ and the set of even counting numbers $\{2, 4, 6, 8, \ldots\}$. Both sets are clearly infinite. Moreover, it seems that the set of counting numbers is about twice as large as the set of even counting numbers. Strangely though, the two sets have the same cardinality: We can establish a bijection - a perfect matching – between the two! Here it is:

Counting Number	Even Counting Number
1	2
2	4
3	6
⋮	⋮
42	84
⋮	⋮

Notice that the mapping here associates each counting number x with the even counting number $2x$. Let's see if it's truly a bijection. Is every counting number associated with a distinct even counting number? Yes, because each pair of counting numbers is matched to two different even numbers. And, is every even counting number matched in this way to some counting number? Yes, because any even counting y number is matched to the counting number $y/2$. So, strangely, these two sets have the same cardinality!

By similar arguments, the set of all integers (the counting numbers, the negatives of the counting numbers, and 0) also has a bijection with the counting numbers, so these two sets also have the same size. Amazingly, even the set of all rational numbers (numbers that are of the form $\frac{p}{q}$ where p and q are integers) has a bijection with the counting numbers. That's truly strange because on the face of it, it seems that there are *way* more

rational numbers than counting numbers. But such is the reality of infinity!

"Larger" Infinities

Any set that has the same cardinality as the counting numbers is said to be *countably infinite*. So, as we noted above, the set of even counting numbers, the set of integers, and the set of all rational numbers are all countably infinite. If that's the case, aren't *all* infinite sets countably infinite?

Surprisingly, the answer is "no"!

The story begins in Germany in the last years of the nineteenth century. A brilliant mathematician name Georg Cantor gave a beautiful argument that shows that the set of real numbers – the set of all numbers, including the rational and irrational numbers - contains a "larger" infinity of elements than the set of counting numbers.

Cantor's proof made many mathematicians in his day very uncomfortable and, in fact, many went to their graves feeling that there must be a problem somewhere in Cantor's logic. Cantor's proof is not only rock-solid but it is one of the most important results in all of mathematics. Moreover, as we'll see shortly, it's the very basis for showing that there exist uncomputable problems.

Cantor's proof is based on the method of *proof by contradiction*. (See the sidebar on proofs by contradiction.) The approach is to assume that something is true (e.g., that the set of real numbers is countably infinite) and derive from that a contradiction (e.g., $1 = 2$ or something equally obviously false). We are therefore forced to conclude that our initial assumption (e.g., that the set of real numbers is countably infinite) must also be false (because if we assume it's true we are led to conclude something that is definitely false).

A Quick Primer on Proofs by Contradiction

In case you haven't seen the technique of *proof by contradiction* before, let's take a brief aside to demonstrate it with a famous mathematical result that goes back to the days of ancient Greece.

The Greeks wondered if all numbers are rational - that is, whether every number can be expressed as the ratio of two integers like $\frac{1}{2}$, $-\frac{3}{7}$, etc. According to legend, Hippasus of Metapontum discovered that $\sqrt{2}$ is not rational - it cannot be written as a ratio of two integers. The ancient Greeks,

also according to legend, treated this result as an official secret, and Hippasus was murdered when he divulged it.

Here's the proof. Assume that $\sqrt{2}$ is rational. Then it can be written as a ratio of two integers, $\frac{p}{q}$. Every fraction can be written in lowest terms (that is, canceling out common factors), so let's assume that $\frac{p}{q}$ is in lowest terms. In particular, that means that p and q cannot *both* be even, since if they were we would have cancelled out the multiples of 2 from each of p and q when putting the fraction in lowest terms.

OK, so now $\sqrt{2} = \frac{p}{q}$. Let's square both sides to get $2 = \frac{p^2}{q^2}$ and now let's multiply both sides by q^2 to get $2q^2 = p^2$. Since $2q^2$ is even, this means that p^2 is even. If p^2 is even, clearly p must be even (since the square of an odd number is always odd!). Since p is even, let's rewrite it as $p = 2\ell$ where ℓ is also an integer. So now, $2q^2 = p^2 = (2\ell)^2 = 4\ell^2$. Since $2q^2 = 4\ell^2$, we can divide both sides by 2 and we get $q^2 = 2\ell^2$. Aha! So q^2 is even! But this means that q is even. That's a contradiction because when we wrote $\sqrt{2} = \frac{p}{q}$ we stipulated that $\frac{p}{q}$ is in lowest terms, and thus p and q could not *both* be even.

We've just established that if $\sqrt{2} = \frac{p}{q}$ in lowest terms then *both* p and q must be even and thus $\frac{p}{q}$ is not in lowest terms. That's like saying if it's raining then it's not raining. In that case, it can't be raining! And in just the same way in our case, $\sqrt{2}$ cannot be written as a fraction.

In a nutshell, a proof by contradiction shows that something is false by first assuming that it's true and then deducing an absurd (namely false) outcome. This forces us to conclude that our assumption was false, thereby proving what we seek to prove!

Cantor not only showed that the real numbers are uncountably infinite, he showed that even the set of real numbers between 0 and 1 are uncountably infinite! Here's how his proof goes: Assume that the set of real numbers between 0 and 1 is countably infinite. (Note that he's setting up the proof by contradiction. He's hoping to eventually conclude that something absurd follows from this assumption, therefore forcing us to conclude that the assumption is false!) Then, there must exist a bijection – a perfect matching –

between the set of counting numbers and the set of real numbers between 0 and 1. We don't know what that bijection looks like, but it would need to match every counting number with a distinct real number between 0 and 1.

A real number between 0 and 1 can be represented by a decimal point followed by an infinite number of digits. For example, the number $.5$ can be represented as $0.5000\ldots$. The number $\frac{1}{3}$ can be represented as $.333\ldots$. The fractional part of π would be $.141592654\ldots$. So a bijection between the counting numbers and the real numbers between 0 and 1 could be represented by a table that looks "something" like this. We say "something" because we don't know what the actual association would be between counting numbers and real numbers would be, except for the fact that it is necessarily a bijection so we must associate a distinct real number with every counting number and every real number must appear somewhere on the right-hand column of the listing.

Counting Number	Real Number
1	$.500000\ldots$
2	$.333333\ldots$
3	$.141592\ldots$
$\vdots$	$\vdots$
42	$.71828\ldots$
$\vdots$	$\vdots$

Now, Cantor says: "Aha! Well now that you have given me the bijection, I'm going to look at that bijection and highlight the first digit of the real number matched with counting number 1, the second digit of the real number matched with counting number 2, the third digit of the real number matched with counting number 3, and so forth, all the way down the list. This is going down the diagonal of the listing of real numbers indicated in boldface in the table below.

Counting Number	**Real Number**
1	$.\mathbf{5}00000\ldots$
2	$.3\mathbf{3}3333\ldots$
3	$.14\mathbf{1}592\ldots$
$\vdots$	$\vdots$
42	$.71828\ldots$
$\vdots$	$\vdots$

"Now," says Cantor, "Watch this! I'm going to write down a new number as follows: My new number differs from the real number matched with 1 by changing its first digit (e.g., changing 5 to 6). Similarly, my new number differs from the real number matched with 2 by changing its second digit (e.g., changing 3 to 4). It differs from the real number matched with 3 by changing its third digit (e.g., changing from 1 to 2), etc. In general, for any counting number n, look at the real number matched to n in your bijection. My number will differ from that real number in the n^{th} digit. For example, for the hypothetical bijection that begins as in the table above, the real number that I construct would begin with $.642\ldots$."

"What's with that number?" you ask. "Well," says Cantor, "You promised me that you had a bijection between the counting numbers and the real numbers between 0 and 1, and that real number that I just constructed is certainly between 0 and 1. So where is it in the table that represents your bijection?" He's got a good point. Where is it? You might argue, "Be patient, it's matched with some very large counting number way down that list." But if so, you must be able to tell Cantor what counting number that is. Perhaps, Cantor's new number is matched with the counting number 10^9. But in that case, by the way that Cantor constructed his new number, it would differ from that number in the billionth digit. That's a problem because we asserted that we had a bijection between the counting numbers and the real numbers between 0 and 1, and therefore every real number in that range, including Cantor's new number, must appear in that bijection.

So, the conclusion is that no matter what bijection we try to establish between the counting numbers and the real numbers between 0 and 1, Cantor can always use his *diagonalization method* (changing the digits along the diagonal) to show that our putative matching fails to be a bijection. In other words, no bijection exists between the counting numbers and the real numbers between 0 and 1 and we have shown that the real numbers are not countably infinite!

A fine point: Our attorneys have advised us to say one more thing here. Note that some real numbers have two different representations when we use an infinite decimal expansion. For example 0.3 is the same as $0.2999\ldots$. Indeed, the problem is with an infinite number of 9's. Every real number whose decimal expansion eventually has an infinite sequence of 9's can be represented by changing the digit before the first of those 9's and then replacing all of the 9's by zeroes. So, it might be that Cantor's new number in his diagonalization proof is actually *not different* from all numbers in our bijection table, it just *looks different* because it has a different representation. The solution to this problem is to always agree to use the "nicer" representation of numbers when we have the choice - that is, never use the representation that has an infinite number of

consecutive 9's. After all, this is just an issue of representing numbers and we can always represent a number that contains an infinite sequence of 9's with its alternate representation! Now, when Cantor diagonalizes, he must be careful not to construct a number with an infinite sequence of consecutive 9's, because we decided to avoid that representation. One easy solution to this is to simply have Cantor avoid ever writing the digit 9 in the new number he constructs. In particular, if he was changing the digit 8 in one of the real numbers in the table, rather than letting him change it to 9 (which might lead to an infinite sequence of 9's), have him change it to any digit other than 8 or 9. This will still give him the contradiction that he seeks, and avoids the legal tangle that our attorneys have been worrying about.

7.3.2 Uncomputable Functions

I wonder how much they billed us for that?

We've just seen that the set of real numbers between 0 and 1 is such a large infinity that its elements can't be matched up perfectly with the counting numbers. But what does this have to do with computer science?

Your "favorite" language is officially Python, but this would work just as well for any programming language.

Our goal is to show that there exist problems that are not solvable by any program. We'll do this using Cantor's diagonalization method. Here's the plan: First we'll show that the set of all programs is countably infinite; that is, there is a bijection between the counting numbers and the set of all programs in your favorite language.

Then, we'll show that the set of problems does not have a bijection with the counting numbers (it's a larger infinite set), and thus there is no matching between programs and problems.

Let's start with the set of programs. A program is, after all, nothing more than a string of symbols that we interpret as a sequence of instructions. Here's our plan: We'll list out all possibles strings in alphabetical order from short ones to longer ones. Each time we list one out, we'll check if it's a valid Python program. If it is, we'll match it to the next available counting number.

There are 256 different symbols that are used in a standard computer symbol set. So, we'll first list the 256 strings of length 1. None of them are valid Python programs. Then we'll move on and generate 256×256 strings of length 2. None of these are valid Python programs. Eventually, we'll come across a valid Python program and we'll match

that with the counting number 1. Then, we'll march onwards, looking for the next valid Python program. That program will be matched with 2, and so forth.

While this process is laborious, it does show that we can find a bijection between counting numbers and Python programs. Every counting number will be matched to a distinct Python program and every Python program will eventually be generated this way, so it will be matched to some counting number. So, there is a bijection between the set of counting numbers and the set of Python programs. The set of Python programs is, therefore, countably infinite.

Now let's count the number of problems. The problem with problems is that there are so many different kinds of them! So, to make our life easier, let's restrict our attention to a special kind of problem where we take a counting number as input and return a Boolean: `True` or `False`. For example, consider the problem of determining if a given counting number is odd. That is, we'd like a function called `odd` that takes any counting number as input and returns `True` if it's odd and `False` otherwise. That's an easy function to write in Python. Here it is:

```
def odd(X):
        return X % 2 == 1
```

A function that takes any counting number as input and returns a Boolean is called a *counting number predicate*. Counting number predicates are obviously a very, very special kind of problem.

It will be convenient to represent counting number predicates as infinite strings of "T" (for `True`) and "F" (for `False`) symbols. We list all the possible inputs to the predicate along the top and, for each such input, we put a "T" if the predicate should return `True` for this input and we put a "F" otherwise. For example, the odd predicate would be represented as shown in Figure 7.2

Inputs to Predicate								
1	2	3	4	5	6	7	...	42
T	F	T	F	T	F	T	...	F

Table 7.2: Representing the odd counting number predicate as an infinite list of "T" and "F" symbols, one for each possible input to the predicate.

Table 7.3 shows a list of several counting number predicates. The first one is the predicate for determining if its input is odd. The next one is the predicate that determines if its input is even, the next one is for primes, etc. Incidentally, all of these predicates have relatively simple Python functions.

	Inputs to Predicate								
Predicate Name	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**...**	**42**
odd	T	F	T	F	T	F	T	...	F
even	F	T	F	T	F	T	F	...	T
prime	F	T	T	F	T	F	T	...	F
true	T	T	T	T	T	T	T	...	T
forty-two	F	F	F	F	F	F	F	...	T
⋮									

Table 7.3: A few sample counting number predicates and the values they return for small non-negative integers.

Now, suppose you claim to have a bijection between the counting numbers and counting number predicates. That is, you have a listing that purportedly perfectly matches the counting numbers to the counting number predicates. Such a bijection might look something like Table 7.4. In this table each row shows a counting number on the left and the predicate with which it is matched on the right. In this example, the counting number 1 is matched to the odd predicate, the counting number 2 is matched to the even predicate, etc. However, this is just an example! Our plan is to show that no matter how we try to attempt to match counting numbers to predicates, the attempt will necessarily fail.

Counting Number	Inputs to Predicate									
	1	**2**	**3**	**4**	**5**	**6**	**7**	...	**42**	...
1	**T**	F	T	F	T	F	T	...	F	...
2	F	**T**	F	T	F	T	F	...	T	...
3	F	F	**F**	T	F	T	F	...	F	...
4	T	T	T	**T**	T	T	T	...	T	...
5	F	F	F	F	**F**	F	F	...	T	...
⋮										
Cantor	F	F	T	F	T	...				

Table 7.4: Applying Cantor diagonalization to the counter number predicates. The table shows an attempted bijection of counting numbers to predicates. Cantor diagonalization is used to construct a predicate that is not in this list.

Once again Cantor reappears to foil your plan. “Aha!” he says. “I can diagonalize your predicates and create one that’s not on your list.

And sure enough, he does exactly that, by defining a predicate that returns values different from everything you listed. How? For the predicate matched to the counting number 1, he makes sure that if that predicate said “T” to 1 then his predicate would say “F” to 1 (and if it said “F” to 1 his predicate would say “T” to 1). So his predicate definitely differs from the predicate matched to the counting number 1. Next, his predicate is designed to differ from the predicate matched to the counting number 2 by ensuring that it says the opposite of that predicate on input 2, and so forth down the diagonal! In this way, his new predicate is different from every predicate in the list. This diagonalization process is illustrated in Table 7.4.

So, Cantor has established that there is at least one predicate that’s not on your list and thus the counting number predicates must be uncountably infinite, just like the real numbers.

What have we shown here? We’ve shown that there are more counting number predicates than counting numbers. Moreover, we showed earlier that the number of programs is equal to the number of counting numbers. So, there are more counting number predicates than programs and thus there must be some counting number predicates for which there exist no programs.

Said another way, we’ve shown that there exist some “uncomputable” problems. That’s

surprising and amazing, but the key words there are “there exist.” We haven’t actually demonstrated a particular problem that is uncomputable. That’s our next mission.

7.4 An Uncomputable Problem

In the last section we proved that there *exist* functions that can’t be computed by any program. The crux of the argument was that there are more problems than programs, and thus there must be some problems with no corresponding programs.

That’s surprising and amazing, but it would be nice to see an example of an actual problem that cannot be solved by a program. That’s precisely what we’ll do in this section!

7.4.1 The Halting Problem

At the beginning of this chapter we noted that it would be very useful to have halt checker program that would take another program (like one that we are working on for a homework assignment) as input and determine whether or not our input program will eventually halt. If the answer is “no”, this would tell us that our program probably has a bug!

To make things just a bit more realistic and interesting, we note that our homework programs usually take some input. For example, here’s a program that we wrote for a homework assignment. (Exactly what the objective of this homework problem was now escapes us!)

```
def homework1 (X):
        if X == "spam":  return homework1(X)
        else: return "That was not spam!"
```

Notice that this program runs forever if the input is `'spam'` and otherwise it halts. (It returns a string and it’s done!)

So, imagine that we now want a function called `haltChecker` that takes *two* inputs: A string containing a program, `P`, and a string, `S`, containing the input that we would like to give that program. The `haltChecker` then returns `True` if program `P` would eventually halt if run on input string `S` and it returns `False` otherwise.

We'd *really* like to have this `haltChecker`. We searched on the Internet and found one for sale! Here's what the advertisement says: "Hey programmers! Have you written programs that seem to run forever when you know that they *should* halt? Don't waste your precious time letting your programs run forever anymore! Our new `haltChecker` takes *any* program `P` and *any* string `S` as input and returns `True` or `False` indicating whether or not program `P` will halt when run on input string `S`. It's only $99.95 and comes in a handsome faux-leather gift box!"

Evidently, you can buy just about anything on the Internet.

"Don't waste your money!"–one of our friends advised us. "We could write the `haltChecker` ourselves by simply having it run the program `P` on string `S` and see what happens!" That's tempting, but do you see the problem? The `haltChecker` must *always* return `True` or `False`. If we let it simply run program `P` on input string `S`, and it happens that `P` runs forever on `S`, then the `haltChecker` also runs forever and doesn't work as promised. So, a `haltChecker` would probably need to somehow examine the contents of the program `P` and the string `S`, perform some kind of analysis, and determine whether `P` will halt on input `S`.

But it's true that we shouldn't waste our money on this product because we're about to show that the `haltChecker` cannot exist. Before doing that, however, our lawyers have advised us to point out that it *is* possible to write a `haltChecker` that works correctly for *some* programs and their input strings, because certain kinds of infinite loops are quite easy to detect. But that is not good enough! We want one that is absolutely positively 100% reliable. It must render a decision, `True` or `False`, for *any* program `P` and input `S` that we give it.

A hypothetical `haltChecker` would need to take two inputs: a program and a string. For simplicity, let's assume that the program is actually given as a string. That is, it's Python code in quotation marks. The `haltChecker` will interpret this string as a program and analyze it (somehow!). So, we might do something like this with our hypothetical `haltChecker`:

```
>>> P = ' def homework1 (X):
                if X == "spam":  return homework1(X)
                else: return "That was not spam!" '
>>> S = 'spam'
>>> haltChecker(P, S)
False
>>> haltChecker(P, 'chocolate')
```

```
True
```

We will show that a `haltChecker` cannot exist using a proof by contradiction again. This is a general and powerful technique for showing that something can't exist.

Here's a metaphorical sketch of what we're about to do. Let's imagine that someone approaches you and tells you, "Hey, I've got this lovely magic necklace with a crystal oracle attached to it. If you ask it any question about the wearer of the necklace that can be answered with `True` or `False`, it will always answer the question correctly. I'll sell it to you for a mere $99.95."

It sounds tempting, but such a necklace cannot exist. To see this, assume by way of contradiction that the necklace does exist. You can force this necklace to "lie" as follows: First you put the necklace on. Then, your friend asks the oracle necklace: "Will my friend accept this handful of jelly beans?" If the oracle necklace says `True` then your friend can ask you "Would you like this handful of jelly beans?" You now answer `False` and the oracle has been caught giving the wrong answer! Similarly, if the oracle necklace says `False` then, when your friend asks you if you would like the jelly beans, you say `True` and you have again caught the oracle in a lie. So, we are forced to conclude that an always accurate oracle necklace cannot exist.

Necklaces? Oracles? Jelly Beans? "You professors need a vacation," we hear you say. Thank you for that, it is indeed almost vacation time, but let's see first how this necklace metaphor bears on the `haltChecker`.

OK then. Here we go! Assume by way of contradiction that there exists a `haltChecker`. Then, we'll place that `haltChecker` function in a file that also contains the program `paradox` below:

```
def paradox(P):
        if haltChecker(P, P):
                while True:
                        print 'Look! I am in an infinite loop!'
        else: return
```

Take a close look at what's happening here. This `paradox` program takes a single string `P` as input. Next, it gives that string as *both* inputs to the `haltChecker`. Is that OK?! Sure! The `haltChecker` takes two strings as input, the first one is interpreted as a Python program and the second one is any old string. So, let's just use `P` as both the program

string for the first input and also as the string for the second input.

Now, let's place the code for the `paradox` function in a string and name that string `P` like this:

```
>>> P = "def paradox(P):
                if haltChecker(P, P):
                        while True:
                                print 'Look! I am in an infinite loop!'
                else: return"
```

Finally, let's give that string `P` as the input to our new `paradox` function:

```
>>> paradox(P)
```

What we're doing here is analogous to you foiling the hypothetical oracle necklace; in this case the hypothetical oracle is the `haltChecker` and you are the invocation `Paradox(P)`.

Let's analyze this carefully to see why. First, notice that this `paradox` program either enters an infinite loop (the `while True` statement loops forever, printing the string `"Look! I am in an infinite loop!"` each time through the loop) or it returns, thereby halting.

Next, notice that when we run the `paradox` function on the input string `P`, we are actually running the program `paradox` on the input string containing the program `paradox`. It then calls `haltChecker(P, P)` which must return `True` or `False`. If `haltChecker(P, P)` returns `True`, it is saying "It is true that the program `P` that you gave me will eventually halt when run on input string `P` ." In this case, it is actually saying, "It is true that the program `paradox` that you gave me will eventually halt when run on the input program `paradox` ." At that point, the `paradox` program enters an infinite loop. In other words, if the `haltChecker` says that the program `paradox` will eventually halt when run on the input `paradox`, then the program `paradox` actually runs forever when given input `paradox`. That's a contradiction; or perhaps we should say, that's a paradox!

Conversely, imagine that `haltChecker(P, P)` returns `False`. That is, it is saying, "The program `P` that you gave me will not halt on input `P` ." But in this case, `P` is `paradox`, so it's actually saying "The program `paradox` will not halt on input `paradox` ." But in that case, we've designed the `paradox` program to halt. Here again, we have a contradiction.

We could easily write the `paradox` program if we only had a `haltChecker`. However, we

just saw that this leads to a contradiction. Thus, we're forced to conclude that our one assumption – namely that a `haltChecker` exists – must be false. In the same way, we concluded that an oracle necklace cannot exist because if it did then we could devise a logical trap just as we have done here!

7.5 Conclusion

In this chapter we've taken a whirlwind tour of two major areas of computer science: complexity theory and computability theory. Complexity theory is the study of the "easy" and "hard" problems - problems for which algorithmic solutions exist but the running time (or memory or other resources) may vary from modest to outrageous. In fact, complexity theory allows us to classify problems into more than just two categorize "easy" and "hard", but into many different categories that ultimately provide deep and surprising insights into what makes some problems solvable by efficient algorithms while others require vastly more time (or memory).

Thanks for reading!

Computability theory explores those problems that are "impossible" to solve, such as the halting problem and many others. In fact, computability theory provides us with some very powerful and general theorems that allow us to quickly identify what problems are uncomputable. Among these theorems is one that says that virtually any property that we would like to test about the behavior of a program is uncomputable. In the case of the halting problem, the behavior that we sought to test was halting. Another behavior we might be interested in testing is that of having a virus that writes values into a certain part of your computer's memory. Results from computability theory tell us that testing an input program for that property is also uncomputable. So, if your boss ever tells you, "your next project is to write a completely reliable virus detector," you might want to read a bit more about computability theory so that you can show your boss that it's not just hard to do that, it's downright impossible.