

CS for Scientists and Engineers

Christine Alvarado
Zachary Dodds
Geoff Kuenning
Ran Libeskind-Hadas

June 18, 2010

This material is based upon work supported by the National Science Foundation under Grant No. CNS-0939149. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation (NSF).

To the Reader

Welcome! This book is a companion to the course “CS for Scientists and Engineers” developed at Harvey Mudd College (HMC) over the last several years. This course, and book, take a unique approach to “Intro CS”.

Our name is Mudd!

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*. We provide a broad view of computer science, including its strong connections to engineering and mathematics and its fundamental importance to the modern practice of all of the sciences. While programming is certainly an important and pervasive element of our approach, we emphasize concepts and problem-solving over syntax and programming-language features.

This approach is evident from the very beginning. In the introductory chapter 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.

Next, in Chapter 3, we pose the question “how does a computer run your recursive program?” We describe the inner workings of a computer from switches to digital logic gates to adders and memory up to a plausible computer. We then program that computer in a simple assembly language, culminating in the implementation of recursive programs.

This may appear to be a topic “switch,” but we feel that it is a “current” approach.

Now that the computer has been demystified and students have a physical representation of what happens “under the hood” of the computer, we move on in Chapter 4 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 5 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 6 poses the question “Are there problems that computers cannot solve?” We provide a gentle but mathematically sound treatment of the theory of uncomputability, ultimately proving that there are many computational problems that are provably impossible to solve on a computer. Rather than using formal models of computation (e.g. Turing Machines), we use Python as our model of computation.

While this course and book are broadly accessible, it is written with scientists, engineers, and mathematicians in mind. The course is required for *all* freshmen at Harvey Mudd College, irrespective of their intended major. Thus, it serves as the first and last course in computer science for many scientists, engineers, and mathematicians and it serves as the introductory course for computer science majors.

This course has been very successful by many measures. A number of papers explaining the theory and assessment of this course have appeared in computer science education conferences and can be found on the Web at [URLHERE](#).

This book is intended to be used with the substantial resources that we have developed for the course and are available on the Web at <http://www.cs.hmc.edu/IntroCS>. These resources include complete lecture slides, a rich collection of weekly assignments, some accompanying software, and documentation.

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!

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

Contents

To the Reader	i
1 Introduction	1
1.1 Computer Science: Realities and Myths	1
1.1.1 Algorithms: Making Pie and Making π	2
1.1.2 Beyond Algorithms	3
1.1.3 The Role of Programming in Computer Science	4
1.2 Picobot	5
1.2.1 The Roomba Problem	6
1.2.2 The global environment and the local surroundings	6
1.2.3 State	7
1.2.4 Think locally, act globally	7
1.2.5 Whatever	11
1.2.6 The Picobot challenge	11
1.2.7 Uncomputable environments	12
2 Functional Programming	13
2.1 Humans + Machines + Aliens = CS	13
2.2 A First Program	14
2.3 Statements and Expressions	18
2.4 Variables: Giving Names To Things	18
2.4.1 Assignment Statements in the 1-2 Prediction Program	20
2.5 Functions	21
2.5.1 <code>return</code> vs. <code>print</code>	22
2.5.2 Functions Can Call Functions!	23
2.5.3 Functions in the 1-2 Prediction Program	25
2.6 Conditional Statements	25
2.6.1 Boolean Expressions (And Alien Special Numbers!)	27
2.6.2 Multiple Conditions	28
2.6.3 Conditionals in the 1-2 Prediction Program	28
2.7 Scope	31
2.8 Recursion	33
2.8.1 Recursion in the 1-2 Prediction Program	35
2.9 Improved Mind Reading	35

3	<i>SuperFunctional</i> Programming	39
3.1	Top-down Human Design	39
3.2	Top-down Program Design	40
3.3	The alien’s protein-folding code	40
3.3.1	Disclaimers	44
3.4	Sequences	45
3.4.1	Strings	45
3.4.2	Lists	48
3.4.3	Operators for lists vs strings	48
3.4.4	Converting between data types	51
3.5	Matters of <code>import</code>	51
3.6	Functions are data, too: <code>reduce</code> , <code>map</code> , and <code>lambda</code>	53
3.6.1	Motivating <code>reduce</code>	53
3.6.2	<code>reduce</code> : a function that takes a function as input	54
3.6.3	Reducing nothing at all?	55
3.6.4	One chain to rule them all	56
3.6.5	A map of <code>map</code>	56
3.6.6	Use it or lose it	57
3.6.7	Functions returning functions	58
3.6.8	The biochemistry, at last!	59
3.6.9	Anonymous functions: <code>lambda</code>	60
3.6.10	And beyond...	61
4	Computer Organization	63
4.1	Introduction to Computer Organization	63
4.2	Representing Information	63
4.2.1	Integers	63
4.2.2	Arithmetic	65
4.2.3	Negative Numbers	66
4.2.4	Fractional Numbers	66
4.2.5	Letters and Strings	67
4.2.6	Structured Information	68
4.3	Logic Circuitry	69
4.3.1	AND, OR, and NOT	69
4.3.2	Making Other Boolean Functions	70
4.3.3	Logic Using Electrical Circuits	72
4.3.4	Computing With Logic	73
4.3.5	Memory	75
4.4	Building a Complete Computer	77
4.4.1	The von Neumann Architecture	78
4.5	HMMM...	81
4.5.1	A Simple HMMM Program	81
4.5.2	Looping	84
4.5.3	Functions	86
4.5.4	Recursion	88
4.5.5	The Complete HMMM Instruction Set	92
4.5.6	A Few Last Words	92

5	Imperative Programming	95
5.1	Motivation: Measuring the Stroop Effect	95
5.2	Repeated Tasks: Loops	100
5.2.1	Recursion vs. Iteration at the Low Level	101
5.2.2	Definite Iteration: <code>for</code> loops	103
5.2.3	How is the control variable used?	105
5.2.4	<i>Accumulating</i> answers	106
5.2.5	Indefinite Iteration: <code>while</code> loops	108
5.2.6	<code>for</code> loops vs. <code>while</code> loops	109
5.2.7	Creating infinite loops on purpose	110
5.2.8	Iteration is efficient	111
5.2.9	Assignment by <i>reference</i>	112
5.2.10	Mutable datatypes can be changed using other names!	114
5.3	Taking advantage of mutable data: Sorting for Stroop	115
5.3.1	Why <code>selectionSort</code> works	117
5.3.2	A swap of a different sort	117
5.3.3	2D Arrays and Nested Loops	118
5.3.4	Dictionaries	121
5.4	Displaying Colored Strings (Randomly)	123
5.5	Putting It All Together: Program Design	125
5.5.1	A program to quantify the Stroop Effect	125
6	OOPs! Object-Oriented Programs	131
6.1	A Rational Solution	132
6.2	Overloading	136
6.3	Printing an Object	138
6.4	A Few More Words on the Subject of Objects	139
6.5	Why are OOPs Important?	141
6.6	Getting Graphical with OOPs	142

CONTENTS

Chapter 1

Introduction

1.1 Computer Science: Realities and Myths

Computer science is to the information revolution what mechanical engineering was to the industrial revolution.

—Robert Keller

Computer Science (CS) is a young and exciting field and one that touches all of us everyday. Strangely, it is also a field that is generally misunderstood. Most educated people have at least a reasonably good idea of what goes on in chemistry or physics, but not so with computer science. We've asked hundreds of bright, educated people what a computer scientist does, and here are some of the common (but way off!) answers we have received:

Computer scientists try to figure out how computers work.

Computer scientists are experts on how to set up, configure, and repair computers.

Computer scientists are experts at programming.

Let's talk about these misperceptions for a minute. First, trying to figure out how computers work is not a good use of time; computers are made by humans and their inner workings are completely understood (at least by some people). Some computer scientists *do* work on the design of the next generation of computers.

Setting up and repairing computers is important, but it's not computer science. You may be surprised to learn that when one of us professors has a malfunctioning computer the only thing that we know how to do is put in a box and send it back for repair! Some computer scientists know how to set up or configure computers just as some musicians know how to tune a piano.

Programming is definitely important and we'll say more about it in a few pages. For now, let's make this observation: Most computer scientists know how to program just as most writers know how to construct grammatical sentences and most astronomers know how to operate a telescope. 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 does not define the discipline.

By the time you've finished this book, and the associated exercises and projects, you will understand what computer scientists do because you will have *done* real computer science. You will have experienced the breadth, complexity, and fascinating variety of this incredible field. You probably will *not* be any better at repairing computers than you are right now.

A formal definition might run: "Computer science is the systematic study of information and computation using that information." Do you feel enlightened?

We could attempt to give a "formal" definition of computer science, but such definitions are inherently so general that they are rarely enlightening. Computer science is a broad discipline with roots in disparate fields that include engineering, mathematics, cognitive science, and even philosophy. Can you imagine a short definition that brings together all of these fields? Neither can we. So instead of giving you a formal definition, we'll begin with some examples of what computer scientists do.

Computer scientists engage in all kinds of activities, ranging from developing new products we use in mobile phones, computers, or web browsers to designing software for life-saving medical devices or air traffic control systems. They invent new techniques for rendering realistic computer graphics in video games and movies. They design robots that can clean a house, drive across a desert, or explore another planet. Usually not the same robot for all three tasks.

The unifying theme is that computer scientists are interested in the *automation of tasks* ranging from artificial intelligence to zoogenesis. Put another way, computer scientists are interested in finding computational techniques or "algorithms" for a wide range of tasks. Next, they write these algorithms as programs, or software, that can be executed on a computer.

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

1.1.1 Algorithms: Making Pie and Making π

Perhaps the oldest known algorithm is the famous *Euclidean algorithm* for finding the greatest common divisor of two positive integers. The algorithm appears in Euclid's writing around 300 B.C.E. but was probably known by other Greek thinkers one or two hundred years earlier. Although Euclid and his Greek friends didn't know it, they were among the first computer scientists!

Algorithms are so fundamental to computer science that they deserve a more complete introduction. *Algorithms* are precise sequences of steps for carrying out a task. How precise? Precise enough that someone (or *something*) with no prior knowledge of the task and with only very basic capabilities could complete it.

Algorithms are commonly compared to recipes. For example, consider the following recipe (algorithm) for making pumpkin pie:

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 in preheated 425-degree oven for 15 minutes.
7. Reduce oven temperature to 350.
8. Bake for 30-40 minutes more, or until set.

Figure 1.1: Approximating π with darts. The area of the square is 4. The area of the circle is πr^2 which is π in this case. The expected fraction of darts that end up in the circle is therefore $\pi/4$. Therefore, if we throw n darts and k land inside the circle, then k/n should be approximately $\pi/4$. In other words, π is approximately $4k/n$.

9. Cool for 2 hours on wire rack.

[cite: Libby’s pumpkin pie recipe—we can use a different one depending on copyright issues.] 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 meters.
2. Inscribe a circle of radius 1 meter (diameter 2 meters) 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.
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 .

Please don’t try this at home.

See Figure 1.1.1 for a visual representation and an explanation of why this algorithm works.

Again, the steps in this algorithm are straightforward. Even if we know nothing about π , if we follow the steps precisely, we will obtain a good estimate for its value. In the case of estimating π , it turns out that the steps can be performed by a computer. Happily, the computer does not actually throw physical darts, but we can simulate this dart throwing process by generating random coordinates where the darts land. Performing this algorithm on a computer is much more efficient (and safer) than doing it by hand. The computer can throw millions of virtual darts in a fraction of a second and will never miss the square, making it considerably safer for your roommate.

1.1.2 Beyond Algorithms

Of course, this distillation of computer science as algorithm design omits many important and fascinating pieces. For example, how do we know if a given task can be automated? It turns out that there are very many useful tasks that we would like to

automate—turn into algorithms and then, ultimately programs—that we can *prove* are not automatable! We’ll see how to write such proofs later in this book.

If a problem can be automated, how do we know that the algorithm that we’ve created is a good one? Usually there are many different algorithms for a given task, and some may be better than others in the sense that they are more accurate, faster, simpler, etc. Designing and evaluating approaches to a computational problem is a creative and challenging endeavor. You’ll get plenty of opportunities throughout this course to design and evaluate solutions to interesting computational problems.

Programs often interact with people. For example, applications that you use on the web are programs. You’ve undoubtedly encountered some confusing ones that elicited head-scratching, teeth-gnashing, or worse. Hopefully, you’ve also encountered web applications that were so easy and natural to use that you barely batted an eyelid. The design of a good user interface is not a coincidence! It’s a rich and important area of computer science and one that you will encounter in this course as well.

Next, there’s the computer itself. Although computers don’t look like much, there is a huge amount of cleverness inside. There are the actual hardware components that ultimately “run” your program and there is some ingenious software that controls the many disparate parts of a computer to make sure that they play nicely together. How does this all work? In this course you will get a chance to design some of the key components of a computer to see how the computer works. Ultimately, this will allow you to understand what happens “under the hood” when a program runs.

Finally, there’s the issue of writing the program. You might think that writing a program is a mindless and mundane task, but nothing could be further from the truth! In fact, the issue of writing programs is so important that we devote the whole next section to it.

1.1.3 The Role of Programming in Computer Science

As we noted earlier, many people believe that computer science and programming are one and the same. Indeed, it is likely that one of the few expectations you had when you picked up this book is that it would help you learn to program.

Of course, you are not wrong in your expectations—you *will* know how to program when you finish reading this book. However, it should be noted that some fine textbooks teach computer science without teaching any programming. You can be a computer scientist without knowing a specific programming language, and, as we will see later, some branches of computer science involve no programming at all.

So can computer science exist without programming? Of course not. Programming is the building block for everything that computer scientists study. It allows us to put our theories and algorithms into practice, both to test them and to apply them to real-world problems.

To illustrate the essential relationship between programming and computer science, let’s consider an analogy: *Computer Science is to programming as culinary arts is to cooking*. Can one be a chef without cooking? Of course, and there are many chefs that do not cook. However, if no one were cooking, what would be the point to having chefs? When someone developed a new recipe, how would they know if it *really was* better than the former recipe? Furthermore, the whole field of culinary arts would never have developed if no one were cooking food in the first place. Finally, and most

Some other books teach programming without really teaching computer science.

importantly, developing recipes without implementing them misses the point entirely. Few people want a good recipe; what they want is a delicious meal.

The same ideas apply to computer science and programming. If there were no programming, there would be no programs that solve useful problems. Few people want to know only about the *idea* of a social network and how one can theoretically stay in touch with friends. What most of us want is Facebook, which is simply a program, even if a very large and complicated one.

In this book we will use the Python programming language to implement algorithms to solve a number of problems. However, because computer science is so much more than just programming, we will also look at “languages” other than Python for programming a computer, including a language named Picobot, introduced in the next section, and a language called HMMM. HMMM is an *assembly language*, which is a very low-level language, directly translatable into the 1’s and 0’s that the computer understands. (We’ll also look at what it means for the computer to “understand,” so don’t worry if that last sentence didn’t make too much sense).

After the description above, you might be tempted to think that programming is a mere mechanical process, in which we translate our algorithms into a form that the computer can interpret. On the contrary, programming, like cooking, is both an art and a science, and mastering that art (and science) is essential to mastering computer science. Writing a program can be an intellectually challenging and exciting undertaking. Once a program is written, it rarely works quite right. Testing your program, finding its “bugs”, and correcting those errors requires some clever detective work. You’ll do all of that here. Let’s get started!

1.2 Picobot

Leap before you look.

—W. H. Auden

We can talk and talk about what computer science is and is not, but 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 or not 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>.

or Bebo or Habbo or
LinkedIn or MySpace or
Orkut....

HMMM is short for the
*Harvey Mudd Miniature
Machine*.

Probably the most
famous series of books in
computer science is a
three volume series titled
“The Art of Computer
Programming” by
Professor Donald Knuth.
Knuth (the “K” is not
silent) is the youngest
person ever to win the
National Medal of
Science. President
Jimmy Carter bestowed
Knuth this honor in
1979.

If your room has no open
floor space, this claim is
“vacuously” true!

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

1.2.1 The Roomba Problem

Or, at least, the “breakout” app that enabled the industry’s first large-scale profits.



An iRobot Roomba. ... especially if they told you that you couldn’t go out with your friends until you were done. 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 actually “see” at all. Instead, it might sense its environment through one of many possible sensors including bump sensors, infrared, camera, laser, etc.

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

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.

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. For one, it can’t even remember which part of the room it has seen and which part it has not.

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 global environment and the local surroundings

Our first task in solving this problem (or any computational problem) is to represent it in a way that the computer can handle. 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 give list the coordinates of the corners of the edges of the objects, and the coordinates of Picobot’s location. While this representation is reasonable, we will actually use a slightly simpler representation.

Whether lawn or sand, the environment is simpler to cover if it is discretized into cells as shown in Figure 1.2. 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); as we mentioned above, it does not know the positions of those obstacles ahead of time. What Picobot can sense are 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 North, then what’s to the East next, then West, and finally South. 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.

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? This information isn't available simply by examining its surroundings. 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, etc. In computer science, we often use “state” to refer to the internal information that describes what a computer is doing.

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.

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.

For example, imagine that Picobot wants to perform the subtask 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).

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 subtask that you would like Picobot to undertake.*

We hope you are currently in an inquisitive state.

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

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,

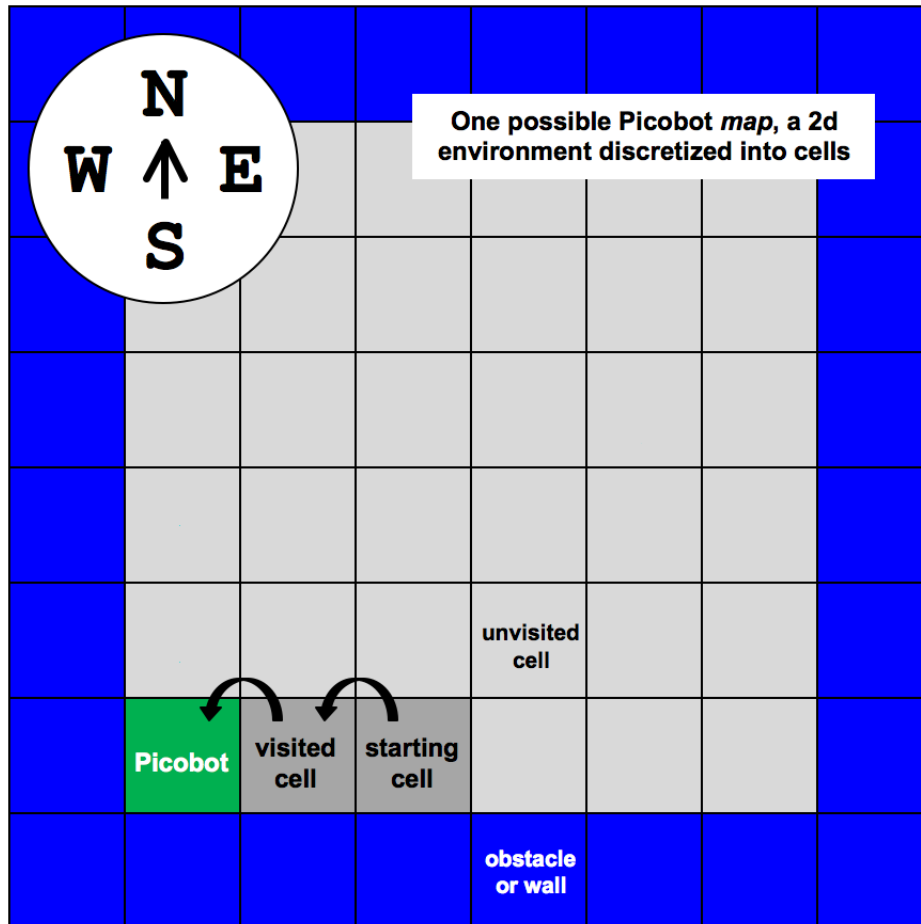


Figure 1.2: There are four types of cells in a Picobot environment, or map: green is Picobot itself, blue cells are obstacles, and gray cells are free space. Picobot can't sense whether a free cell has been visited or not (dark or light gray), but it can sense whether each of its four immediate neighbors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

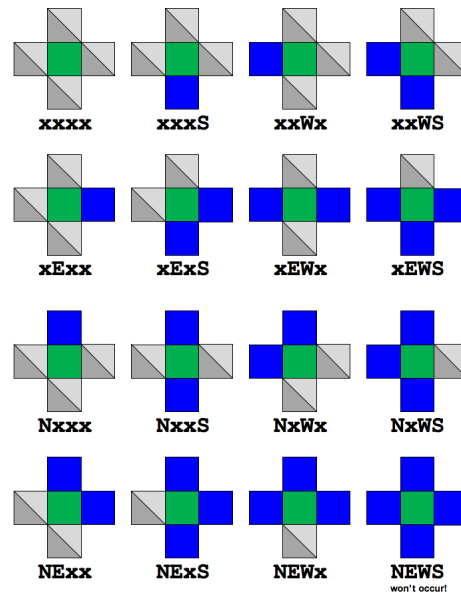


Figure 1.3: There are sixteen possible surrounding strings for Picobot. The one in which Picobot is completely enclosed will not occur in our simulator!

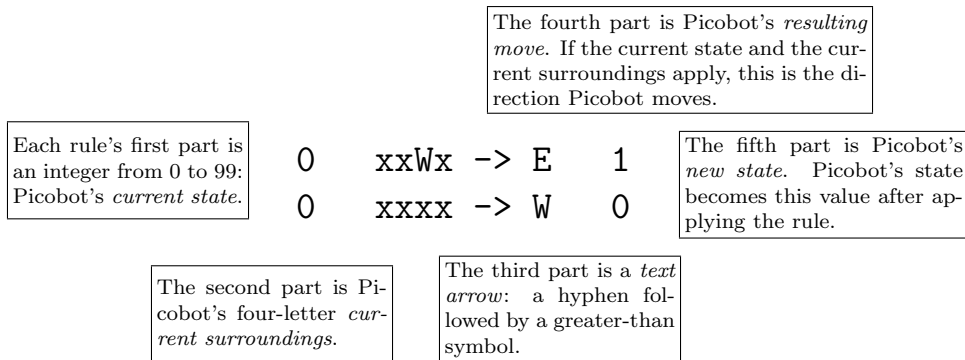


Figure 1.4: The five parts of two Picobot rules. One useful way to interpret the idea of state is to attribute a distinct intention to each state. With these two rules, Picobot's initial state (state 0) represents "go west as far as possible."

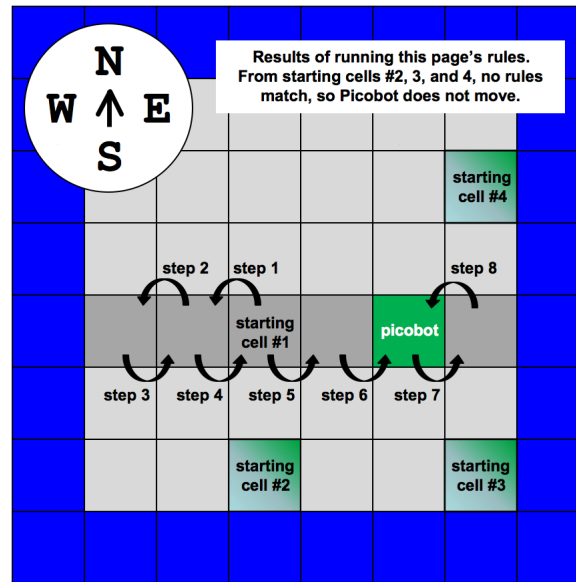


Figure 1.5: The result of running Picobot with this section’s four rules.

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

Go west, young Picobot!

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.

Remember that Picobot begins its mission in state 0.

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. If more than one rule applies, Picobot just uses the first rule from the top, so the other rules are ignored. Picobot can not sense whether or not a cell has been visited. This limitation is quite realistic: the Roomba, for example, does not know whether a region has already been cleaned.

How does Picobot choose which rule to use? At each step, Picobot examines the list of rules that you’ve written from the **top down**, looking for the first 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 Picobot’s current surroundings. The first rule that matches is the one that Picobot uses. It then moves in the direction specified by that rule and enters the state dictated by that rule.

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” subtask, we can specify two rules that will make state 1 be the “go east” subtask:

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—the Picobot web site contain these four rules by default. The web site also starts Picobot at a randomly selected free cell. Note, however, that

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

Figure 1.6: 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!

if Picobot starts along a wall, no rules match and it does not move! We will remedy this defect in the next section.

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 obstacles 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 * expresses this “don’t care” attitude.* Table 1.6’s rules use the wildcard to direct Picobot to forever clear the east-west row in which it starts.

In case you care, the star character “” is also known as an asterisk. Keep it away from your world records!

1.2.6 The Picobot challenge

Table 1.6’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 Figure 1.2’s and 1.5’s empty environment, starting from any free cell. 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 start by altering the rules in Figure 1.6 so that they side-step into a neighboring row after clearing the current one.

Once you develop a set of rules that will always clear the empty environment, go further—try developing sets of rules that will always clear more intricate environments, such as those shown in Figure 1.7. Indeed, you may consider using *as few* rules as possible. A ruleset that completely clears the empty square environment from any starting cell will necessarily contain *at least five rules*. We can quantify different environments’ complexity using this minimum-number-of-rules criterion, as Figure 1.7 describes.

This back-and-forth
ot nwonk saw euqinhcet
ancient Greek ox-drivers,
ti dellac ohw
“boustrophedon.” Text
lacissalc emos ni
manuscripts is known to
.nrettap emas eht wohs

Zach: I’d like to add rules and states in that figure—also, I’d be interested in knowing if there are cases where the minimum number of rules and the minimum number of states are incompatible. P.S. All of the rule numbers stated here are completely made up—but I’m happy to work on that at some point... Question from Ran: Is the five

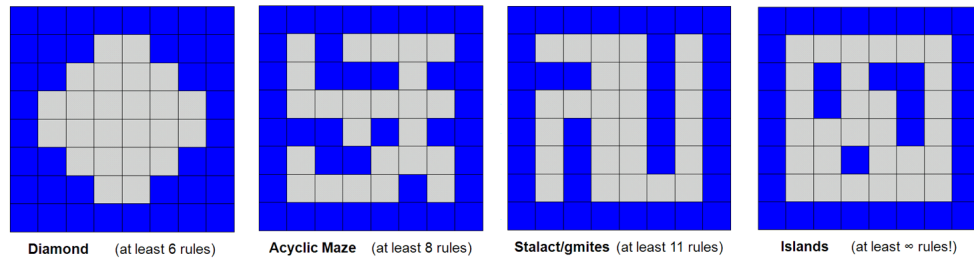


Figure 1.7: Some Picobot environments and the number of rules needed for Picobot to be able to visit every cell in the environment. *Geoff: Zach, I found the infinity symbol unreadable in the Islands environment. I'd suggest breaking this into four sub-figures without text, and then we can use L^AT_EX to insert the captions and make sure they're set in a consistent typeface.*

rule minimum hard to show? Also, the number of rules is a reasonable measure of complexity but the number of states is also a good measure.

1.2.7 Uncomputable environments

Picobot's computational capabilities aren't enough to guarantee coverage of all environments, e.g., those with islands with concavities (that is, "inlets"). By adding the ability to drop, perceive, and pick up "pebbles" along the way, all environments can be completely traversed. The Picobot web site defines extensions to the language that let you do just that.

The fact that computational challenges as elementary as Picobot lead us to *provably unsolvable problems* suggests that computation—and computers—are far from omnipotent. When thinking computationally, it is important to know if a problem might be altogether out of computer science's reach!

Chapter 2

Functional Programming

If it's a good idea, go ahead and do it...—Admiral Grace Hopper

2.1 Humans + Machines + Aliens = CS

In the spirit of Grace Hopper, one of the pioneers of computer science, this chapter plunges directly into CS by showing how a computer can read the human mind. All right, so we've *slightly* overstated the plan. Here's the “true” story...

An alien has landed on planet Earth and wishes to study the predictability of humans. To that end, the alien sets up a booth on a street corner offering free lemonade in exchange for participation in an experiment. Thirsty and curious humans stop and are asked to repeatedly type the number “1” or “2” on the alien's computer. Before each keypress, the alien's computer program tries to predict the human's next number. The human is asked to enter a number and the computer indicates whether or not it guessed correctly. When the human wants to quit (indicated by pressing the number “3”), the computer reports the number of times it guessed correctly and the total number of rounds that were played. Our goal in this chapter is to experiment with programs for this problem.

Before we dive in, let's pause for a moment to consider the utility of being able to predict human behavior. Perfect prediction is probably impossible. But even a limited ability to predict human behaviors has many practical applications. For example, you've benefitted from Google's ability to predict your query even before you've finished typing it. Much of science and engineering is about making good predictions, not just of an individual human's behavior but also of other complex systems. For example, meteorologists attempt to predict the weather, energy engineers attempt to predict electricity consumption, and economists attempt to predict financial trends. All of these problems are difficult and require sophisticated algorithms that are implemented in computer programs.

All right, back to our alien and the human “1-2 prediction” problem. If humans are truly random, any program the alien writes will do no better than chance. Similarly, if the program uses no information at all to try to predict what the user will guess (i.e., the computer simply guesses randomly too), the program will again do no better



Evidently, the authors have *lost* their minds!

This is probably a good thing!

than chance. The alien wants a program that uses some sort of simple rules that will hopefully do better than chance.



Demonstrating that Python is universally popular!

2.2 A First Program

The alien has traveled from the other end of the universe to Earth with a short Python program for this task. Download the program at [URLHERE](#) and try it out! When running this program, you might try inputting 1s and 2s as fast as you can (following each number by pressing “return”) for several dozen iterations. You can stop the program by entering the number 3.

If you’re not totally impressed by this program’s ability to predict your input, we (uhh, we mean the alien) won’t be offended. In this chapter we will learn enough Python to understand how the alien’s program works and then you’ll be invited to make improvements to it.

Take a look at the program in Figure 2.1. Some of it may make sense while much of it may seem truly “alien.” Let’s start with a high-level look at what this program is doing just to get a rough sense of what’s happening here. In the rest of this chapter we’ll look at the program in considerably closer detail.

The program begins with the *comment* “Prediction program, version 1.0” and then another comment indicating the author of this program on the next line. In Python, any line that begins with #, called the “pound symbol”, is interpreted by the program as a *comment*. A comment is nothing more than a note to the reader of the program. It will not appear when the program is run, but it’s very useful for the programmer to write a note to herself or to others who might wish to modify the program later.

Next, the program has two *functions*, one called `getInput` and the other called `play`. As we’ll see in a moment, a function is a module in a program; its purpose is to solve some particular task. In Python, a function always begins with the word `def`. The job of `getInput` is to ask the user to type in a number and to give that number back to another part of the program. The job of the `play` function is to run one round of the experiment.

At the beginning of each of those functions, you will see some text. It begins and ends with three double quotation marks. This text is called a *docstring* which stands for “documentation string”. A “string” in computer science parlance is just a piece of text. A docstring is a bit like a comment with the following difference: While comments are intended for programmers, docstrings are intended for the users of the program. Once you load in this program, you can type `help(getInput)` or `help(play)` and it will display the docstring for that function. *You should always provide a docstring for every function that you write.*

To better understand the notion of a function, consider the following completely uncontrived scenario: Imagine that you work at a unicycle rental shop where your function is to staff the front desk. Let’s call you the “front desk function”. When a customer arrives, you ask them for their height and take their money. Then, based on how much they have paid and their height, you shout to the bloke working in the back “I need a basic short unicycle” or “I need a fancy tall mountain unicycle”, etc. Let’s call that bloke the “unicycle providing function”. That function returns a unicycle to

We are gearing up to make a fortune on a chain of unicycle rental shops.


```
# Prediction program, version 1.0
# Written by: Alien

def getInput():
    """Asks the user to enter the number 1, 2, or 3 and returns the
       user's input."""

    userReply = input( "Enter 1 or 2 (or 3 to quit): " )
    if userReply == 1 or userReply == 2 or userReply == 3:
        return userReply
    else:
        return getInput()

def play(thisGuess, correct, total):
    """Plays a round of the game.  thisGuess is the computer's next
       guess, correct is the the number of times the computer has
       guessed correctly, total is the the total number of times that
       the user has played."""

    userChoice = getInput()
    if userChoice == 3 :
        print "I predicted", correct, "correctly"
        print "in", total, "rounds of play"
        return
    elif thisGuess == userChoice:
        nextGuess = 3 - thisGuess
        play(nextGuess, correct+1, total+1)
        return
    else:
        nextGuess = thisGuess
        play(nextGuess, correct, total+1)
        return

# here is the start of the overall program...
print "I will try to predict your numbers!"
play(1, 0, 0 )
```

Figure 2.1: The alien's first attempt at a prediction program.

you. You then take that unicycle, add a helmet, and give those items to the customer.

A function, in this case, is a person who performs a very specific job. Analogously, in a program, a function is a piece of the program that performs a particular task. It's not strictly necessary to have multiple functions, each of which performs a specific task; we could simply have one giant program that does it all. Similarly, a unicycle shop could have one employee that does all of the work. One advantage of having functions though is that it divides up the work into logical chunks, or modules, that make it much easier to figure out what's going on and much easier to make changes later. The principle of *modularization* was borrowed from the field of engineering and it is a key idea in computer science.

Notice that in our unicycle rental shop example, the functions get input. For example, you get money and information on the customer's height. The bloke in the back gets an order from you. Similarly, these functions return something to whoever "invoked" or "called" them. The bloke in the back returns a unicycle to you, since you asked for it. You return a unicycle and a helmet to the customer because they asked for it. Similarly, functions in a program may get some input when they are invoked and may return some output.

In our Python program in Figure 2.1, the `getInput` function is a bit funny in that it gets no input when it is invoked. This is indicated by the open and closed parentheses in `getInput()`. Instead, this function asks the user to type in a number. Then, if that number is a 1, 2, or 3, it returns that value to whoever invoked that function. Otherwise, the human typed in an expected input and the function asks for input again. Take a look at the statement `return userReply`. This is where the number is being returned to whoever called `getInput`.

Wait a second! We said that `getInput` doesn't get any input when it is invoked but it *does* ask the human for some input. How does that make any sense? Here's an analogy from our unicycle shop. The bloke in the back decides that he doesn't like to have you call him with the customer's information. He just want you to buzz him to indicate that a client has arrived. When you press the buzzer now, you're not passing any information but you are invoking the bloke. The bloke now comes out, chats with the client, and then returns a unicycle to you so that you can complete the transaction. This is precisely what's happening in the `getInput` function. It gets no input when it's called by you, but it is certainly capable of interacting with the outside world, getting the data that it needs, and returning something to you.

The `play` function, in contrast, gets three inputs which we have named `thisGuess`, `correct`, `total`. The input `thisGuess` will be the computer's prediction of the human's next input. So, the value of this input will be either 1 or 2. The input `correct` will be used to keep track of the number of times the computer has guessed correctly. Finally, the input `total` will keep track of the number of rounds that we've played; that is, the number of times the human has entered a number 1 or 2.

Although the `play` function may still seem rather weird, notice the first thing that it does in the line

```
userChoice = getInput()
```

This line is calling the `getInput()` function. This is akin to you calling the bloke in the back of the unicycle shop. The `getInput` function is going to return a value of

1, 2, or 3. This value is then being assigned to the variable `userChoice`. The `play` function then decides what to do next. The line

```
if (userChoice == 3):
```

is saying “if the value of `userChoice` is 3, then I will do the sequence of steps that are indented below me.” In this case, it prints some messages on the screen in the first two lines and then, in the third line it returns to whoever invoked this function. Note that in contrast to the `getInput` function, the `play` function returns nothing. It just says “OK, I’m done!”

When Python is asked to run this program, it sees the comment line at the top and ignores it (that’s intended for programmers to read). It then sees the two function definitions and makes a note of them, but doesn’t do anything noticeable yet. Finally, at the very bottom of the program are the lines

```
print "I will try to predict your numbers!"
play( 1, 0, 0)
```

When Python sees this, it says “Aha! You aren’t *defining* a function here—you’re actually telling me to *do* something!” The first line prints a message (whatever is in quotes). The next line calls (or “invokes”) the `play` function. Fortunately, we’ve already defined that `play` function, so Python understands what we’re asking of it.

Python now invokes (or “calls”) the `play` function with the three inputs 1, 0, 0. The 1 goes into the first slot in the `play` function, namely `thisGuess`. This is going to be the first guess that the computer makes. The next input, 0, goes into the `correct` input in the `play` function. This the number of times that the computer has guessed the human’s answer correctly. Since the human hasn’t chosen a number yet, 0 is the appropriate initial value. Finally, the last input, 0, is the total number of rounds that the human has played. This too is initially 0.

Whew! Again, don’t worry if much of this program is still a mystery encased in an enigma and wrapped in a riddle. Most of the rest of this chapter methodically explains the Python features used here (and some other features too!) so that every last drop of this program will make sense (and you’ll be able to modify it to make it better).

If you’re feeling brave, we’d urge you to just pause here and play a bit with the program. Try changing just one small part of the program. Then save the change and run the program to see what it does. For example, you might just try changing the message that is printed when the program starts. Then, you might try to add a line in the `play` function that prints the computer’s next guess immediately after the human has entered its next number. Finally, see if you can figure out the method (the algorithm) that this program is using to choose its next number. (*Hint*: $3 - 1 = 2$ and $3 - 2 = 1$. You knew that, but see where 3 appears in the program.) Based on this knowledge, run the program and see if you—as the human user—can make it predict your input very badly. If you’re feeling even more brave, try changing the prediction method used by the computer to some other method. It needn’t be better, just try something different.

A human who performs this experiment a few times might quickly realize this! We should fix this later.

Don’t worry, it’s impossible to break the computer this way!



Experimentation is fun and eye-opening!

2.3 Statements and Expressions

The Python interpreter offers us a direct way to interact with Python, even before we've written any programs. If you type a piece of data at Python's `>>>` prompt, the language evaluates what you've typed and prints the resulting value. For example, try typing `3*14` in the Python interpreter and press the Return key. Python evaluates this expression and returns a value, in this case 42.

A Python input like `3*14` is called an *expression* because it expresses a value and this value can be used later for other purposes. Now take a look at the input `print 3*14`. In this line, the expression `3*14` is evaluated and then python gives this value to `print` which displays the value on the screen. The input `print 3*14` is called a *statement*. It is a command to do something but ultimately this command has no value.

Whoa, Nellie! Hold up here. When we typed `3*14` it printed 42 and when we typed `print 3*14` it also printed 42. They seem the same, but they are not. The difference is that you can *use* the value of an expression in another expression. You cannot use the result of a statement, because a statement does not have a value. Try adding 5 to the two inputs noted above: typing `5 + 4*13` yields 47. Typing `5 + print 4*13` yields a `SyntaxError`. In the latter case, Python saw the 5 followed by the + and said "Aha! This is an expression. You want me to evaluate it. OK then, what should I add to 5?" Then it saw the statement `print 4*13` and it said "Hang on! This has no numerical value—it's not an expression! How do I add 5 to a command that is asking me to print?"

So, the basic building blocks of all computer programs are *statements*, which are pieces of code that tell the computer to perform a specific task. Statements often contain expressions in them, just as the statement `print 3*14` contains the expression `3*14`. In Python, statements are separated from each other by putting each statement on a separate line.

Many of the lines in the 1-2 prediction program correspond to individual statements, including the following:

```
print "I will try to predict your numbers!"  
  
and  
  
userChoice = getInput()
```

Again, don't worry if you don't understand the precise meaning of these statements yet. We'll go over their meaning shortly.

2.4 Variables: Giving Names To Things

One of the most important aspects of programs is the ability to store values for later use. The expression `3*14` is lovely, but how do we use it later without recomputing it? In the 1-2 prediction program, for example, we need to store lots of things: the computer's guess, the human user's answer, the number of times we guessed correctly, etc. *Variables* give us a way to name values so that we can store these values away for later use.

To give a value a name (i.e. store it in a variable), we first cook up a name for the variable and then use an *assignment statement* to give it a value. Let's define a variable named `yourNumber` and let's give it a value.

```
>>> yourNumber = 2
```

Now, let's say that you change your mind and you want to give this variable a new value.

```
>>> yourNumber = 3*14
```

Notice that this is a statement! It has no value. However, the statement involves evaluating the expression `3*14`, which Python determines to be `42` and then it assigns this value to the variable `yourNumber`. Now we can ask Python to return the value stored in this variable as shown below:

```
>>> yourNumber
42
```

Since you're certainly familiar with the notion of equality from mathematics, the assignment statement may feel a bit weird. It is not stating an equality, it is performing an action. It is best understood from right to left. On the right-hand-side of the equals sign is an expression that yields a value (in our example, the number `42`). On the left, is the name of a variable, or storage location, for that value. The equals sign then binds the name on the left to the value on the right.

This binding allows us to use the stored value in future statements. For example:

```
>>> yourNumber = 4*13
>>> myNumber = yourNumber - 1
```

Again, Python evaluates the statement on the right-hand-side (henceforth denoted "RHS") of the equals sign and binds it to the name on the left-hand-side (henceforth "LHS"). So in this case, `myNumber` will get the value `41`. All the names on the RHS are used for their values; only the value of the variable on the LHS will change. This idea is extremely important, and it allows us to write statements such as:

```
>>> yourNumber = 4*13
>>> yourNumber = yourNumber - 1
```

In math, the second statement would be nonsense because it looks like we are asserting that the LHS and RHS are equal. How can a variable equal its own value minus 1? However, in Python this assignment statement is perfectly legal because it is defining an action rather than asserting a mathematical truth. Python first evaluates the expression on the RHS of the equals sign (whose value is `41` in this case) and then assigns it to the variable on the LHS. It doesn't matter to Python that this variable appears on both the LHS and RHS, because it evaluates the RHS before it even considers what is on the left.

One important "safety tip" about variables: You cannot try to access their value before you assign them a value. For example, the statement:



It has "value" in that it's useful!

What's in a Variable Name?

Python allows almost any sequence of alphanumeric characters `a-z`, `A-Z`, and `0-9` to be a variable name. Even punctuation and other weird symbols can appear in a variable name. One restriction: it does have to start with a letter and it cannot have any blanks in it. However, it can be something like `value1` or `my_favorite_number`.

```
>>> x
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'x' is not defined
```

produces an error (that's the longwinded stuff beginning with "Traceback blah, blah, blah") because `x` does not yet have a value. Similarly (and more subtly),

```
>>> x = x + 1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'x' is not defined
```

also produces an error because the RHS of the assignment statement is evaluated first. In this case, the variable `x` has no value when Python tries to evaluate the expression on the RHS, so Python complains.

2.4.1 Assignment Statements in the 1-2 Prediction Program

Finally, let's consider how assignment statements and variables are used in the 1-2 prediction program. Take a look, for example, at the following line in the `play` function in Figure 2.1:

```
nextGuess = 3 - thisGuess
```

This statement is evaluating the expression `3 - thisGuess` and assigning that value to a variable that we've named `nextGuess`.

Notice that Python, like any programming language, is very finicky about syntax. For example, in the line

```
print "I predicted", correct, "correctly"
```

The word `correct` is never printed. Instead, Python prints the *value* of the variable `correct`. On the other hand, the words `I predicted` are printed literally (right before the value that `correct` names) because they are in quotation marks and similarly the word `correctly` is printed afterwards because it too is in quotation marks.

Notice that commas can be used to print a sequence of different things. Had this line been

```
print "I predicted", "correct", "correctly"
```

the result would be

```
I predicted correct correctly
```

which would have been incorrect (grammatically at least)!

Editors, IDLE, and More

We haven't written this yet. In the future, this sidebar will talk about editors and the IDLE environment.

2.5 Functions

You've undoubtedly seen the idea of a function in math. For example, consider the function $f(x) = x^{100}$. We'll call this function the "googolization" function because a googol is 10^{100} . (It's true! You can Google the word "googol" to verify this.) Exponentiation in Python is performed using `**`, so 42^{100} would be written as `42**100` in Python. Type this into the Python interpreter to see the result.

Figure 2.2 shows one way we could define the googolization function in Python:

```
def f(x):
    """ Googolization function. Takes input x and returns x**100. """
    result = x**100
    return result
```

Figure 2.2: One way to implement the googolization function.

The Python *keyword* `def` indicates that we are defining a function. The first line of the function definition is called the function *signature* and has the following parts:

1. The keyword `def`, indicating to Python that we are defining a function.
2. The name of the function. In this case it is `f` but the name could have been more interesting. The naming rules for functions are the same as the naming rules for variables.
3. The list of names of input variables (also known as *arguments* to the function) that this function should accept. This list of names is given in parentheses and the inputs are separated by parentheses. There can be zero inputs (in which case we just have open and closed parentheses with nothing inside), one input (as in this example, where the input is name `x`), or more.
4. The line ends with a colon.

We could type this function right into the interpreter, but it's much more convenient to define programs in an editor where we can easily modify them and save them for later use. See the sidebar *Editors, IDLE, and More*. Once we have this function defined, we can run it with many different input values. Try typing `f(42)`, `f(-1)`, etc. to see for yourself.

Returning to our googolization function above, notice that there are two statements, each on a separate line and indented from the function signature. Python is big on indentation and the indentation indicates that these statements belong to this function. In this case, the line `result = x**100` is an assignment statement that



Googol eyes?!

Wow, that value is astronomical!

A keyword is a word that Python recognizes as part of its language.



So I can have arguments with my functions!?

computes the value of the expression `x**100` and assigns that value to a variable that we've named `result`. In the next statement, this value is “returned” or “outputted”. The keyword `return` tells Python to hand this value back to whoever happened to call the function originally. If you type `f(42)` in the Python interpreter then *you* are calling that function, so the value will be returned to you and displayed. (Technically, the Python interpreter called the function and thus the result is returned to the Python interpreter where it is displayed, but this is a nuance.)

At this point, you may be wondering “what’s the difference between `return` and `print` ?” That’s a great question, and we’ll tackle it shortly. For now, think about it this way: `return` gives you back a value that you can use later. In contrast, `print` just displays something on the screen and that’s it.

Before we end this section, let’s write the googolization function in a slightly different way as shown in Figure 2.3. In this implementation, we have a single statement. This statement first evaluates the expression `x**100` and then immediately returns it. This has the same effect as the previous two-line version above, but skips the step of making a variable to name the result.

```
def f1( x ):
    """ Another Googolization function. """
    return x**100
```

Figure 2.3: Another way to implement the googolization function.

Regardless of how we wrote the googolization function, it’s important to note that if the function returns a value, that value can be used like any other value. So, for example, consider typing

```
>>> big = f(42)
```

As usual, Python first evaluates the RHS of the expression. This involves calling the function `f` with input 42. The function `f` does its business and returns some value. Now, Python takes this value and assigns it to the variable `big`. Python is really treating `f(42)` as an expression. Expressions need to be evaluated before they can be used. In this case, evaluation of the expression requires asking the function `f` to compute and return its value. Nothing is printed here! You could type `big` at the Python prompt if you wanted to see its value now.

2.5.1 `return` vs. `print`

We’ve briefly mentioned the difference between `return` and `print`, but let’s take a closer look. At the Python interpreter’s prompt, `return` and `print` can indeed seem like they’re doing the same thing. To see the difference, consider the two functions and sequence of commands shown in Figure 2.4.

It appears that `doubleReturn` and `doublePrint` are doing the same thing. However, the `return` statement and the `print` statements are, in fact, very different. The `print` statement simply displays the value of its expression at the console. No value is returned by the `doublePrint` function, so the function call `doublePrint(20)` *has*


```

def doubleReturn( x ):
    """ doubleReturn takes a single input x and returns 2x. """
    return 2*x

def doublePrint( x ):
    """ doublePrint takes a single input x and prints 2x. """
    print 2*x

>>> doubleReturn( 20 )
40

>>> doublePrint( 20 )
40

```

Figure 2.4: return vs. print

no value at all (to be more precise, that call has the Python default value of `None`). The `return` statement’s value, on the other hand, *becomes the value of the original function call!* That is, the original function call is an *expression* whose value is conveyed by the `return` statement. As with any expression, that function call can be used in further computations:

```

>>> 2 + doubleReturn( 20 )
42

>>> 2 + doublePrint( 20 )
40
TypeError: unsupported operand type(s) for +: 'int' and 'NoneType'

```

In the latter example, note that `doublePrint` did its job! That is, 40 was printed out to the console. However, because `doublePrint` does not `return` anything, the statement `doublePrint(20)` has no value (or, rather, has the value `None`). Thus, Python’s attempt to add the int 2 to the valueless `doublePrint(20)` results in an error.

Neither `print` nor `return` is preferable over the other—they simply accomplish very different things.

2.5.2 Functions Can Call Functions!

Check this out! Functions can call other functions. Consider the function `humongous` in Figure 2.5. This function takes two inputs, which we’ve creatively named `input1` and `input2`. In the first line inside the function, we call the function `f` with input `input1`. When `humongous` calls function `f`, think of it as passing a baton to `f`. Now, `humongous` pauses and waits for `f` to do its business.

When the function `f` gets to its `return` statement, it means that the game is over for `f`. It now passes back the baton to the place where it was called. In addition, it

The word “humongous” is believed to have been invented in 1968 as college slang.

passes back whatever value follows the keyword `return`. (Think of this value as having been glued on to the baton). In the case of `f`, this is the input that `f` received raised to the power 100.

In this example, `humongous` now gets back the baton and resumes where it left off. It takes the value that was glued on to the baton and uses that value in place of `f(input1)`. This value is then multiplied to the value stored in `input2` and then assigned to variable `z`. Finally, this function returns the value `z*z`. That is, `humongous` is now returning the baton to whoever called it and it's gluing its value onto the baton for us to use.

Aside from the fact that this function is highly contrived and almost certainly useless, it *does* demonstrate the fact that a function can call another function. It also demonstrates the following key idea: When a function, like `humongous`, is called, it gets a baton that allows it to compute. If `humongous` calls another function, then `humongous` remembers where it was, passes the baton to the function being called, and waits for the baton to return. When it returns, `humongous` resumes computing where it left off, using the value that was returned (if there was one).

```
def humongous(input1, input2):
    """humongous takes inputs input1 and input2."""
    z = f(input1)*input2
    return z*z
```

Figure 2.5: The `humongous` function calls the `googolization` function.

“Ginormous” became an official word in Merriam-Webster’s Collegiate Dictionary in 2007.

Just for kicks, consider the function `ginormous` in Figure 2.6.

```
def ginormous(input):
    """ginormous takes a single input and computes a strange
       function."""
    a = humongous(42, input)
    b = a**2
    return b
```

Figure 2.6: The `ginormous` function calls the `humongous` function.

Imagine that we invoke the `ginormous` function at the prompt as follows:

```
>>> ginormous(100)
```

Take a moment to write down the sequence of actions that will transpire to make sure that you are comfortable with the idea of the function `ginormous` calling the function `humongous` which in turn calls the function `f`.

Takeaway message: *When a function is called, Python keeps track of where it was at the time of the function call. When the function returns, Python resumes computing exactly where it left off before the function call.*

2.5.3 Functions in the 1-2 Prediction Program

Let's go back for a minute and take a look at the 1-2 prediction program in Figure 2.1. Notice that there are two functions there, one named `getInput` and one named `play`. While we haven't yet examined some of the Python features used in these functions, note that `play` begins by calling the function `getInput`. The function `getInput` begins by calling the function `input`. Where is the function `input` defined?

It turns out that `input` is a function that is built-in to Python. There are many such functions and we'll meet more of them soon. The `input` function takes as input a string: A sequence of letters between quotation marks. It prints that string and then waits for you to type in a number. That number is the value that `input` returns. In the first line of our `getInput` function, the `input` function is called. The value that it returns is then assigned to the value `userReply`. The `getInput` function does a bit more stuff and eventually returns a value. Where does this returned value actually go? To the exact point where the `getInput` function was called, namely the first line in the `play` function. That returned value is then assigned to the variable named `userChoice` and the `play` function continues doing its business.

2.6 Conditional Statements

There is a famous problem in mathematics called the “ $3n + 1$ ” problem. It goes like this. Consider a function that takes an integer n as input. If the integer is odd, the function outputs the value $3n + 1$. If the integer is even, the function outputs the value $\frac{n}{2}$. The problem, which is really a conjecture, states that if we start with any positive integer n and repeatedly iterate applying 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 the input 10 we get the output 5. Applying the function to the input 5 we get 16, applying the function to 16 we get 8, applying the function to 8 we get 4, then 2, then 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 Erdős stated that “Mathematics is not yet ready for such problems.”

Let's write the function in Python and then you can experiment with it! An implementation is shown in Figure 2.7.

Before we look at this in detail, notice the expression `n % 2 == 0`. What is `%` and what is `==`? The expression `n % 2` means the remainder when `n` is divided by 2. Mathematicians pronounce this “`n mod 2`”. If `n` is even, its remainder when divided by 2 will be 0. (We could also ask for the remainder when `n` is divided by 42 with expression `x % 42`, but that's not useful just now!)

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 RHS of `=` to the variable on the LHS. The syntax `==` is doing something quite different. It evaluates the expressions on its LHS and RHS and determines whether or not they have the same value. This kind



My guess is that two equal signs means “very equal”.

```
def collatz( n ):
    """ collatz takes a single integer input n and
    returns the collatz function on n. """

    if x % 2 == 0:
        return 3n+1
    else:
        return n/2
```

Figure 2.7: The $3n + 1$ function in Python.

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`. An expression that has a value of `True` or `False` is called a *Boolean expression*.

Back to our `collatz` function. This function 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 $3n + 1$. On the other hand, if the Boolean expression that was tested had 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 `n/2`.

It turns out that `else` statements are not strictly required and sometimes we can do without them. For example, take a look at a slight modification of our `collatz` program shown in Figure 2.8. If `n` is even, this function computes $3n+1$ and returns

```
def collatz1(n):
    """ Another implementation of collatz """

    if n % 2 == 0:
        return 3n+1
    return n/2
```

Figure 2.8: A second version of the $3n + 1$ function.

that value. Returning that value causes the function to end. Thus, if `x` 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` statement. This is the line `return n/2`. So, we see that this version of the function behaves just like the first version with the `else` clause. Forty-one out of forty-two computer scientists that were surveyed advocate using the `else` version simply because it is easier to read and understand.



What’s all this nonsense with 42?! Do you humans believe that it’s the answer to the ultimate question of life, the universe, and everything!?

2.6.1 Boolean Expressions (And Alien Special Numbers!)

We hope that you’ve found this quick introduction to Python edifying so far, but we sense that you are concerned about the alien and its desire to improve the 1-2 prediction program. We’re getting there, but just to make sure that the alien hasn’t been forgotten, let us pause here to share a true story.

Our alien comes from a planet where certain integers are deemed “special”. A number is special if it is between 42 and 142 (including those numbers) or it is any positive integer that is not a multiple of 7. Our alien became a famous celebrity on its planet for writing a Python function that takes as input any number and determines whether or not it is special. The alien’s `special` program is shown in Figure 2.9.



On *some* planets, programmers are celebrities!

```
def special( number ):
    """ Determines if the single input is special."""

    if 42 <= number <= 142 or number > 0 and not number % 7 == 0:
        print "That number is special!"
        return
    else:
        print "That is a silly boring number"
        return
```

Figure 2.9: The alien’s “special” program.

The expression in the `if` statement is particularly interesting. First, note that the subexpression `42 <= number <= 142` is a Boolean expression that is `True` precisely when 42 is less than or equal to (`<=`) `number` and `number` is less than or equal to 142. In addition to `<=` and `==`, Python also has *comparison operators* `<`, `>`, `>=` (greater than or equal to), and `!=` (not equal to).

Next, we see the use of Python’s `or`, `and`, and `not` operators. The subexpression `not number % 7 == 0` is `True` precisely when `number % 7 == 0` is `False`. In other words, `not` negates (or “flips”) `True` into `False` and vice versa. We could have replaced `not number % 7 == 0` with `number % 7 != 0`.

Notice that our alien’s `special` function prints something and then returns. Since there is no expression immediately following the word `return`, in this case the function is just returning the baton to whoever called it (saying “I’m done! You can continue doing whatever you were doing when you called me.”) but it’s not returning any value for later use.

You’re certainly familiar with the concept of “order of operations” from math. It arises here too. We want

```
42 <= number <= 142 or number > 0 and not number % 7 == 0
```

to be `True` when `number` is 42 (since that number is between 42 and 142 and thus the condition for specialness is satisfied). However, what if Python understood the expression above to mean “`number` must be between 42 and 142 or it must be greater than 0 and, in addition, the `number` must not be a multiple of 7?” Then 42 would not be special since it is a multiple of 7.

The good news is that, in Python, `or` has lower precedence than `and` (which in turn has lower precedence than `not`). So, the expression above will be interpreted as

```
42 <= number <= 142 OR
number > 0 and not number % 7 == 0
```

rather than

```
42 <= number <= 142 or number > 0 AND
not number % 7 == 0
```

If we had wanted (for some weird reason) to force Python to use the latter interpretation, we could use parentheses to force the order of operations as in

```
if (42 <= number <= 142 or number > 0) and (not number % 7 == 0):
```

In fact, when we use `or`, `and`, and `not` in Boolean expressions, it's advisable to use parentheses even if Python will interpret your intention correctly without parentheses. The parentheses make the program much easier for others (and even for yourself) to understand. Forty-one out of forty-two computer scientists advocate this as well.

2.6.2 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 `else`. Did we mention that in addition to "special" numbers, our alien's culture has something called "extra special numbers" as well as "super special numbers"? It's true! The number 42 is super special. The number 10^{100} (a googol) is extra special. Then there are the regular old special numbers as we've already described above. Figure 2.10 shows the alien's program for determining the level of specialness of a number. The `elif` statements are pronounced "else if". This program, starts off checking if `number` is 42, in which case it is super special and we return. Else, if `number` is a googol then the number is extra special and we return. Else if the number is special we report that. If all previous options fail, we report that the number is silly and boring.

In general, we can have as many `elif` conditions as we wish following an `if`. Then, at the end we can have zero or one `else` statement, but not more than one!

Finally, what would have happened if we had rearranged the order of the lines as in Figure 2.11? Something bad happens here. Do you see what it is? In some cases, the order in which we test our conditions is important, particularly when more than one condition may be `True` for our data.

2.6.3 Conditionals in the 1-2 Prediction Program

Take a look back at our alien's 1-2 prediction program. We've talked about the variables and the functions there. Now take a look at the conditional statements. In the `getInput` function, there is an `if` statement that is testing if the user's input, `userReply`, is either 1, 2, or 3. If it is, the value is returned. Otherwise, something weird happens: The function `getInput` is called. This is odd! The function `getInput` is calling *itself*! This is called *recursion* and we'll talk about that very soon. However,



I can vouch for
this.

```
def superExtraSpecial( number ):
    """Determines the level of specialness of its single input."""

    if number == 42:
        print "That number is super special!"
        return
    elif number == 10**100:
        print "That number is extra special!"
        return
    elif (42 <= number <= 142) or (number > 0 and not number % 7 == 0):
        print "That number is special!"
        return
    else:
        print "That is a silly boring number"
        return
```

Figure 2.10: A program for determining special, extra special, and super special numbers.

```
def superExtraSpecial( number ):
    """Attempts to determine the specialness level of its single
    input."""

    if (42 <= number <= 142) or (number > 0 and not number % 7 == 0):
        print "That number is special!"
    elif number == 42:
        print "That number is super special!"
        return
    elif number == 10**100:
        print "That number is extra special!"
        return
    else:
        print "That is a silly boring number"
        return
```

Figure 2.11: This program doesn't quite do what it's supposed to do, due to the order in which we are checking numbers.

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!

That 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.

you might have guessed what is happening here: The input was not what we expected (not a 1, 2, or 3), so we are invoking the `getInput` function again, in essence to give the user a chance to try again.

Now check out the `play` function. It begins by calling `getInput` which (assuming that the human user eventually enters a 1, 2, or 3) will eventually return a 1, 2, or 3. This value is then assigned to the variable `userChoice`. Then, the `if` statement checks to see if `userChoice` is 3. If it is, the human player wants to end the experiment so the program prints some information on what transpired in this experiment and returns. Else, if the program's prediction, `thisGuess`, is equal to the human user's guess, `userChoice`. If so, it computes its next guess and calls the `play` function to play again. (That's recursion again and we'll talk about that soon.) Else, the guess must have been incorrect and the program uses the same guess as its next guess and plays again.


```

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

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

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

>>> demo( 13 )
42

```

Figure 2.12: These functions use the same variable names!

2.7 Scope

We promised we would talk about recursion but we’ve consulted with our lawyers and they told us we could sneak in a short section here 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.

We want to talk about scope here. Take a look at the code in Figure 2.12. How did Python arrive at 42 as its answer in this case? 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 effect the value of `x` in another function?

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 lock boxes that it can use to store “precious” commodities. These lock boxes have the special feature that their doors are on top and they are stackable. Figure 2.13 shows an artists rendition of a stack of lock boxes. We’ve put nice little windows in the boxes so that you can see (but not tinker) with what’s inside. Notice that only the box on the top of the stack has its door accessible.

Lock 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 in Figure 2.12. When the call `demo(13)` is made, the value 13 is passed into the `demo` function’s input variable 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 RHS of the statement `r = f(x+6) + x`. However, `demo` wants to ensure that the “precious”

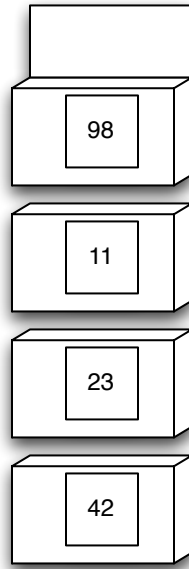


Figure 2.13: A stack of lock boxes. Only the box on top has its door accessible.

value of its variable x , currently, 13, is preserved. It 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 lock box for safe-keeping. In this case, there is a variable called x and Python stores its value, 13. This is shown in Figure 2.14(a).

When the function f is called, it gets the input $13 + 6 = 19$. The value 19 is going into f 's input variable x . Now x has 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 lock 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 lock box for safe-keeping. This lock box is stacked on top of the previous box as shown in Figure 2.14(b). Now function g gets its input of 18 in its input variable of x . It computes $10 + 18 = 28$ and returns that value. When we return to function f , that function immediately finds the lock 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 2.14(c). Notice that 19 is the value that's printed in the line `print x` in the second line of f . Now x is changed to 1. Finally f returns $28 + 1 = 29$ to the `demo` function. At this point `demo` finds its lock box at the top of the stack, opens it, restores its original value of x to 13, and tosses the lock box. This value is now used in the rest of computation, and the `demo` returns the value $29 + 13 = 42$. Whew!

This stack of lock boxes is called the *stack*. In practice it is implemented using the computer's memory rather than steel boxes with cute doors and windows, but we like the metaphor. We'll revisit stacks in subsequent chapters.

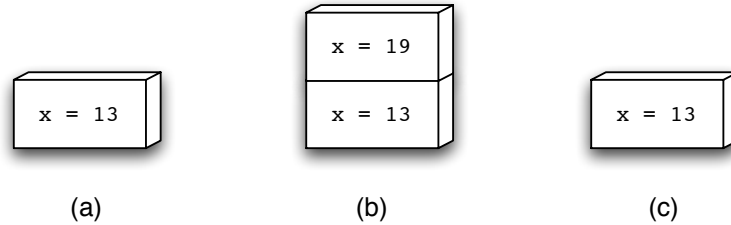


Figure 2.14: The use of the stack when `demo(13)` is evaluated. (a) `demo` places its value of `x` on the stack. (b) `f` places its value of `x` in a box on the top of the stack. (c) The stack after `g` is done and `t` has retrieved its value of `x`

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 Recursion

We haven’t forgotten about our alien. In fact, the alien is sitting at its street-side booth and there are lots of people lined up waiting for free lemonade and a chance to participate in the 1-2 prediction experiment. At the moment there are, let’s see, 42 people in line. The alien muses to itself, “I wonder how many different ways I could arrange those 42 people.” You may know the answer, it’s $42 \times 41 \times 40 \dots \times 3 \times 2 \times 1$, also known as “42 factorial” and written $42!$ (There are 42 people that could be chosen to be the 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 an extra 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* because it expresses the problem in terms of a smaller version of the same problem.

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. Figure 2.15 shows the program. You might try running this program to see what happens.

Amazing! But how does this work? Let’s take a look at what happens when we run `factorial(3)`. We’ll explain it step-by-step and summarize the process in Figure 2.16. 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 input 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 better put my precious belongings in a lock box on the stack for safekeeping so that I



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

```
def factorial(n):
    """ Recursive function for computing factorial of n. """

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

Figure 2.15: A recursive factorial program.

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 lock box for later retrieval. Then, it calls `factorial(2)`. Now `factorial(2)` runs. Notice 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 lock 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 and it’s done. 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 lock 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 computing. It computes `2 * 1`, assigns that value 2 to the variable `result` and returns that result.

That result is returned to the place that `factorial(2)` was called. That was in `factorial(3)`, which now goes to the top of the stack, opens the lock 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, voilá, we have our answer!

We’ll see a lot more recursion in the next chapter. Before we go back and take a look at how recursion was used in our 1-2 prediction program, let’s just examine why we even had the `if n == 1:` statement in our factorial program. What if we had omitted this? If you’re not sure, try omitting it and running the program without the `if` part.

The test `if n == 1` is called a *base case*. (It’s similar to a base case in mathematical induction. In fact, you may have noticed that recursion and induction are very similar in spirit.) Without it, the program has no way of knowing when to stop. Base cases are critically important in writing correct recursive functions.

It is normal for recursion to make your head spin at first. It becomes second nature over time and we’ll see lots more of it in the next chapter. In the meantime, you might like this quip from a former student of ours: “To understand recursion, you must first

understand recursion.”

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!*

2.8.1 Recursion in the 1-2 Prediction Program

Our alien’s 1-2 prediction program in Figure 2.1 uses recursion in a somewhat less “fancy” way than in the factorial function that we just looked at. In the 1-2 prediction program, recursion is really being used just to repeat a process. Let’s start by looking at the `getInput` function. This function takes no input and returns the number entered by the human user. If the input was 1, 2, or 3, `getInput` returns that value. If not, the user typed something wrong, so we call `getInput` recursively to simply ask the user to try again. We’ll loop here until the user finally types in a 1, 2, or 3.

It’s worth taking a moment to ponder what happens if we call `getInput` and the user types in the number 4. Let’s call this the *first* call to `getInput`, just for reference. The first call of `getInput` now calls `getInput` again. Let’s call this the *second* call to `getInput`. If the human user types in a 2 now, that 2 gets returned. But notice that it’s returned to whoever called the *second* call to `getInput`. It was the first call to `getInput` that made that call, so it gets the number 2 back. Now, the first call to `getInput` resumes where it left off. Where was that? It was in the line `return getInput()`. The value of `getInput()` in that line is now 2, so the first call to `getInput` returns that value of 2. Now, we’re done with recursion. The point here, is that the recursive calls can stack or “wind” up. However, when the user finally types in a good input, that input is passed back to the earlier recursive calls as they return and “unwind”, eventually returning that good user input all the way back for us to use.

Finally, let’s take a look at the recursion in the `play` function. This function has three input variables. The first input variable, `thisGuess`, is the value that the function will use as its current guess. The second input, `correct`, is the number of times that the computer’s guess was correct. The third input, `total`, is the total number of times that we’ve played.

Notice when we call this function initially, we call it with `thisGuess` equal to 1 (2 would have worked just as well!), `correct` equal to 0, and `total` equal to 0. In the event that the our guess was the same as the user’s choice, we choose a new value for our next guess (we chose the “other” value so that if the user typed a 1 this time then we will guess a 2 next time and vice versa). We call `play` recursively with this new guess and with the values of `correct` and `total` incremented by 1. On the other hand, if we guessed wrong, the function keeps the next guess the same as the current guess and calls `play` recursively with the `total` incremented by 1. This use of recursion is purely for the purpose of looping.

It may seem loopy, but it’s actually quite sensible.

2.9 Improved Mind Reading

Wow! You’ve just seen a good chunk of the foundations of programming. This is a great time to pause and try out some of the ideas in your own programs. One

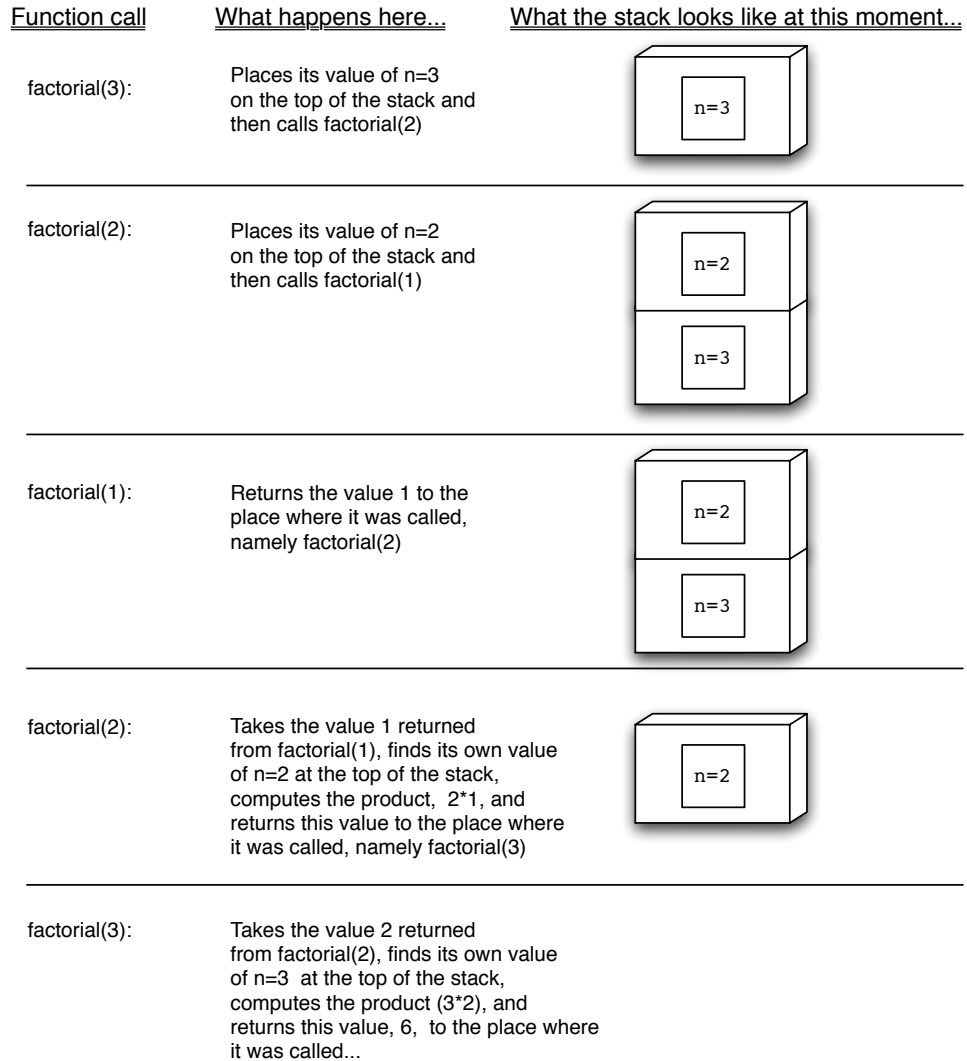


Figure 2.16: What happens when we invoke `factorial(3)`.

place to start is to try modifying the alien's 1-2 prediction program. For example, our prediction algorithm was pretty crude: Start guessing 1. If we guessed right, choose the other possible guess next time. If we guessed wrong, stick with the same guess next time.

There are more sophisticated algorithms that we could try. For example, imagine that we kept track of the last three numbers that the human entered (when we start the program, we might just fake it and pretend that the last three numbers were 1's). Then, our algorithm would look at what number was entered most often in the last three times and use that number as its next guess.

How could we implement this in Python? One possibility is to modify the `play` function so that it takes three extra arguments as input: The number that the human entered three plays ago, two plays ago, and at the previous play. You might want to try this or some other strategy. Get your friends to play and see if one algorithm seems to do better than another.

In the next chapter, we'll see some more advanced features of Python and some more computational problem-solving approaches. These will allow you to add even more sophisticated features to your 1-2 prediction program and to write entirely different and fun programs. In the meantime, congratulations on becoming a functional programmer!

Chapter 3

SuperFunctional Programming

The most dangerous phrase in the language is, *We've always done it this way.*

—Admiral Grace Hopper

3.1 Top-down Human Design

At a loss to explain why humans might be more predictable than machines at choosing a “random” sequence of numbers, our alien plans further investigations into what controls human behavior.

Caffeine seemed a strong first hypothesis as humans’ controlling agent. After all, when the alien switched from free lemonade to free coffee for its experiments, participation skyrocketed! The resulting input data did seem “shakier,” however, and Starbucks’ continued threats of an anticompetitive-practices lawsuit convinced the alien to look elsewhere.

“Well, people are controlled by their subsystems: the muscles, brains, nerves, and all of the other components that make them so unusual,” the alien figured—reasoning *top-down* about the humans observed so far. “But those subsystems are simply collections of cells—what controls those cells’ behavior?” A quick and (relatively) painless “scan” revealed that it *was* a set of molecules that ran the show after all: proteins. Each cell produced long chains of amino acids—called proteins—at astonishing rates.

Even more astonishing was that those one-dimensional chains worked by *folding up* into all the machinery the cells needed for their trillion-strong conspiracy to act human. Understanding proteins, the alien figured, would be the key to understanding people.

“Creating a 1d protein-chain is no problem,” the alien noted—humans have that down pat. But how will I know *which* 1d chains fold into 3d as desirable molecular machines instead of disastrous ones? For that, the alien realizes that it needs another program: one that simulates the folding process. With a folding simulator, the function of the 3d folded protein could be predicted from the original 1d protein-chain.

Aliens, it turns out, aren’t modular at all: they’re just one, large cell.

Indeed, protein folding is the “Holy Grail” of biochemistry—and it remains unsolved by humans or aliens. . .

3.2 Top-down Program Design

Or *zeroth* approximation

A protein-folding program is more complex than the alien’s 1-2 prediction. So, the alien decides to write a first approximation by carefully planning what it will do. First, the overall task is decomposed into its subtasks; those subtasks are further analyzed, in turn, all the way to the level of Python’s functions, it’s fundamental building blocks.

Picture of a three-link chain with angles.

Although the alien does muse hungrily about exactly what those flavors might be

How will a folding program work? First it will set-up the data, defining a 1d list of angles to represent the 1d chain of “protein parts.” Specifically, that data will be the angles between consecutive protein parts. For the time being, the alien *abstracts away* details like the fact that “protein parts” are called amino acids, and that amino acids happen to come in 20 different flavors. With the representation in place, the alien imagines a top-level function, named `fold`, that takes as input the original list of 1d protein parts and then folds it up!

Unfortunately, “folds it up!” isn’t a Python keyword. After filing a bug report with Python’s author for this oversight, our alien decides to break the task “fold it up!” into smaller parts:

- First, the function will create a list of all of the partly-folded chains that are immediate neighbors of the current input chain.
- Next, the function will winnow, or *reduce*, that list of many neighboring chains to the single best neighbor. Biochemists’ best neighbors are the laziest! Thus, the neighboring chain with the smallest total energy will remain.
- Finally, the whole process repeats: the new, lower-energy angles become the input to the next call to fold. En route, an interface can display the molecule either graphically or textually.
- When all neighboring chains all require more energy than the original input to fold, the folding is done. A stable state has been reached, at least until you add heat—or caffeine!

Nature is lazy, but hides it well!

But even the individual steps of the design above require refinement before they’re “Pythonable.” How could we find all the immediate neighbors of a particular chain (step one, above)? For simplicity, these neighbors could be all of the chains whose internal angles are one degree away from the input chain’s. The task of creating all of these lists of angles further decomposes into changing only the first angle and delegating subsequent changes to a recursive call.

Taking a step back, this elaboration is called *top-down* design. At first, a “big idea” is articulated. That idea is then fleshed out into constituent subtasks. Those subtasks, too, receive more and more detailed analysis—until the pieces are simple enough to express in a language like Python.

3.3 The alien’s protein-folding code

Listing 3.1, below, shows the alien’s final effort, a file named `SuperFunFold.py`, which is available from [URLHERE](#). You’ll need the helper file `graphics.py`, too. Notice that, with Python, top-down design leads to code that is perhaps best read from the bottom up! We next look at the parts of this code in more detail.

Listing 3.1: Protein-folding code

```

# The alien's protein-folding program, SuperFunFold.py
# This requires graphics.py in the same "folder", as well.
#
# Remember, scoffing at aliens == bad karma!

import graphics # be sure to have graphics.py
import math, random, time

def pairwise_energy(dist):
    """pairwise_energy
        This is where the bio/chem appears (or should)
        input: a single value, the distance between two
              pieces (peptides) of the molecule
        output: the energy required to maintain that
              distance
    """
    # "Steric" energy is required to hold atoms or molecules
    # so close that their volumes overlap. We use an
    # exponential:
    steric_energy = math.exp((2*20)/dist)
    # "Van der Waals" forces pull molecules to a "preferred"
    # mutual distance of 60. Here, quadratic. More formal
    # estimates are **6 !
    van_der_waals_energy = abs(dist-60)**2
    # we return the sum of these two energies
    return steric_energy + van_der_waals_energy

def dist_from(point):
    """dist_from
        one input: a list of [x,y] coordinates (a point)
        one output: a _function_ that computes distances
                  to the input point!
    """
    x1, y1 = point # "unpack" point into its two
                  # coordinates

    # Note that we are defining f INSIDE dist_from!
    def f(point2):
        x2, y2 = point2
        dist = math.hypot(x1-x2, y1-y2) # built in to
                                        # the math module

        return dist

    return f

```

```
def all_pairwise_distances(points):
    """all_pairwise_distances
        input: a list of lists of two elements (points)
        output: a list of all pairwise distances among
            the input points
    """
    if len(points) < 2: return []

    first = points[0]
    rest = points[1:]

    dists_without_first = all_pairwise_distances(rest)
    dists_with_first = map(dist_from(first), rest)

    return dists_with_first + dists_without_first

def compute_E(angles):
    """compute_E
        input: a list of bond angles
        output: the energy (E) required to maintain those
            angles
    """
    points = graphics.compute_coordinates(angles)
    pairwise_distances = all_pairwise_distances(points)
    E = sum(map(pairwise_energy, pairwise_distances))
    return E

def all_neighbors(angles):
    """all_neighbors
        input: a list of angles
        output: a list of lists of angles, with each angle
            1 degree away from its corresponding
            input angle
    """
    if angles == []: return [[]]

    a = angles[0]
    rest_of_angles = all_neighbors(angles[1:])

    return map(lambda R: [a-1]+R, rest_of_angles) + \
           map(lambda R: [a+1]+R, rest_of_angles)
```

```
def smaller_E(angles1, angles2):
    """smaller_E
        two inputs: both are lists of bond angles within a
                    molecule
        one output: the list of bond angles with smaller
                    energy (E)
    """
    if compute_E(angles1) < compute_E(angles2):
        return angles1
    else:
        return angles2

def fold(old_angles):
    """fold
        input: a list of angles among the elements in the
               "chain"
        output: None, but each call to fold pushes the
               chain into a lower-energy state (until a
               minimum is reached)
    """
    ALL_angles = all_neighbors(old_angles)

    new_angles = reduce(smaller_E, ALL_angles, old_angles)

    if new_angles != old_angles:
        graphics.updateChain(new_angles) # update graphics
        time.sleep(0.05) # pause for 0.05 seconds
        fold(new_angles) # and keep going...

    return

# The program starts here...
raw_input("Hit enter when you're ready to fold...")

# a list of initial angles between the pieces of the protein
initial_angles = [-11.37, 8.51, 15.05, -3.96, -26.87, -25.85,
                  25.89]

# set up the graphics
graphics.createChain(initial_angles)

# run the folding
fold(initial_angles)

# when fold returns, we're finished!
print "I fold, therefore I am."
```

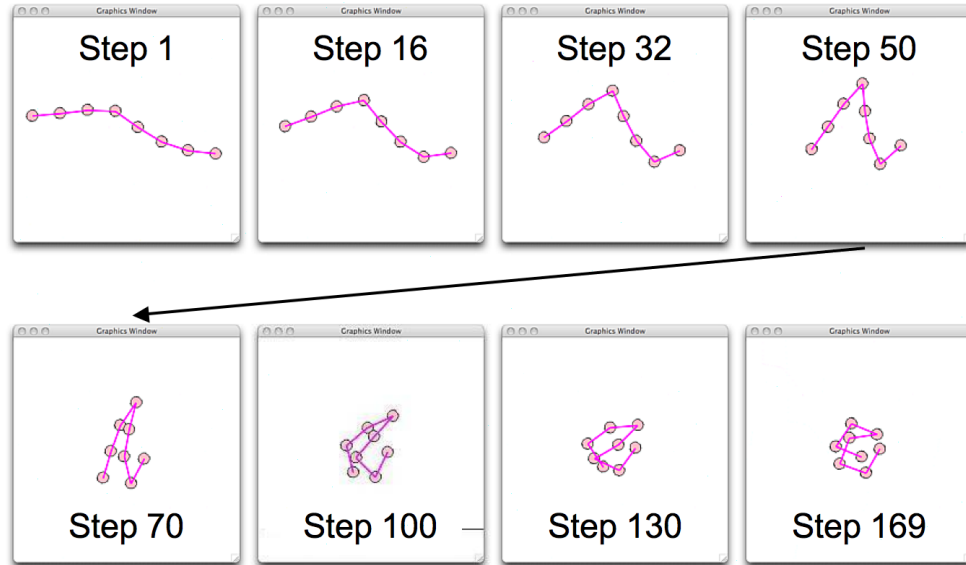


Figure 3.1: Eight snapshots of the protein-folding animation. The code is in Listing 3.1.

3.3.1 Disclaimers

Admittedly, this example abstracts away enough details that biochemists wouldn't bother to mock it. The details we've thrown away about molecule shape, interaction with the surroundings, the factors contributing to the energy landscape, and atomic and quantum effects could fill volumes. Yet, here, they are encapsulated in a single place: the topmost function `pairwise_energy`. A run of this folding simulation resulted in the snapshots below:

In fact, they do!

It's a *really* short video!

In contrast, a video of a few nanoseconds of a better modeled, 3d-rendered, and more accurate folding of a 1d chain into a 3d protein appears at <http://www.youtube.com/watch?v=fvB03TqJ6FE>.

But the abstraction has its advantages: for one, we have abstracted away Python's 2d graphics and some coordinate geometry in the `graphics` module. Thus, it becomes easy to improve the visualization: chapter 5 presents a 3d graphics module, `vPython`, that could be used to “add volume” to our folding simulation.

Another advantage of the abstraction is that it isn't too specialized to be able to adapt to other tasks. For example, by breaking the chain's bonds and adapting the `pairwise_energy` function to represent gravitational attraction, the same computational scaffold results in an *n*-body simulation. Figure 3.2 shows snapshots of a three-body simulation; now, the circles represent macroscopic interactions among physical objects:

It's also at [URLHERE](#).

By the end of this chapter, you'll be able to convert the biochemical simulation into this physical simulation—or write it from scratch, for that matter! This *n*-body simulation solves a problem old enough that Newton struggled over—and that remains



Figure 3.2: Eight snapshots of a three-body dynamic simulation. Gravity is modeled in a vacuum of the non-Roomba sort. Note-I should probably make this 64 snapshots, each 1/8 the size...

analytically intractable even today!

3.4 Sequences

So, how does the folding program work? First and foremost, it needs to handle ordered collections of data. These are the lists of angles under consideration. The line

```
initial_angles = [-11.37, 8.51, 15.05, -3.96, -26.87, -25.85, 25.89]
```

uses the variable name `initial_angles` to represent a list of seven floating-point elements. Lists are one type of sequence datatype Python supports. This list represents the starting state of the internal angles among a protein's chain. A simplified version appears in 3.3.

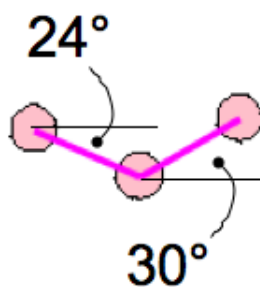
Strings, used in the 1-2 prediction program are a different type of built-in sequences. We take a look at strings and lists in detail below.

3.4.1 Strings

Python's sequences use *delimiters* to indicate beginning and end of the contained elements. As you saw in the 1-2 prediction program, strings can use double-quotes as delimiters, e.g., `"this is a string"`. Single-quotes, as in `'this is a string'` are also OK. The language does not distinguish between these versions, so that

```
>>> "this is a string" == 'this is a string'
True
```

chain represented
by the list [24, 30]



all segments are 1 unit in length

Figure 3.3: Example of the list representation of angles for our linked chains.

However, a string must open and close with the *same* delimiter; otherwise, you will get `SyntaxError`.

```
''' Python also has triple-quoted strings
    These are used as docstrings and can
    contain multiple lines. They can be
    triple-double-quoted, too.

    Ordinary strings must be typed as one line.
'''
```

The empty string

As the above examples show, strings contain strings. In fact, Python's strings may *only* contain strings—or nothing at all. The “empty string” is `""` or `''`.

A `len` function is available to return the length of a string:

```
>>> s = ''
>>> len(s)
0

>>> s = 'Harvey Mudd College'
>>> len(s)
19
```

Note that spaces count: they are strings like any other. Be careful, because the string of a single space is *not* the empty string:

```
>>> " " == ""
False
```

Ultimately, the individual characters of strings are represented as sequences of bits in a machine's memory (see section 4.1). As far as Python is concerned, if a string has any elements, all of those elements are themselves strings.

Indices and slicing

Square brackets *index* into a string, yielding the value of the element at the index's position. Nonnegative indices count from the left, starting from 0; negative indices count from the right. Thus,

```
"abcdefg"[0] == "a"
```

and

```
"abcdefg"[-1] == "g"
```

It possible to extract substrings, too, using both a left and right endpoint separated by a colon. This is appropriately termed *slicing* the string. For example,

```
"abcdefg"[2:4] == "cd"
```

Figure with all of the indexing and slicing examples---we have a slide with them all.

Figure 3.4: caption here

Note that the left endpoint is included, but the right endpoint is not! It is even possible to include a third colon-separated value, which indicates a step size, i.e., how often to include characters in the resulting slice, so that

```
"abcdefg"[0:5:2] == "ace"
```

There are many conventions that govern indexing and slicing: figure 3.5 provides examples that cover most of them. Take a look at each expression and make sure you understand how it works. The three most important conventions to keep in mind as you manipulate strings are

- Counting elements from left-to-right starts from zero. That is, strings—and all sequences—in Python are *zero-indexed*. Remember that `"string"[1] == "t"`!
- Negative indices count from right-to-left, starting from -1 instead of 0. negative step sizes move from right-to-left.
- If indices are omitted in a slice, they default to the front or end of the string. Thus, `"crest"[1:] == "rest"` and `"crest"[:] == "crest"`!

This full-sequence slice turns out to be useful because it *copies* the data before returning it!

3.4.2 Lists


Python's lists, on the other hand, are “strings” that can contain any type of element at all. Lists are delimited by square brackets; two or more elements are separated by commas. For example, `[7,8,9]` is a list of three integers, `[]` is the empty list, and `["jan",31]` is a list of two elements of different types. Indexing and slicing—and all of the conventions they carry—work for lists just as they work for strings. Figure 3.6 gives an opportunity to review those capabilities in the context of lists.

One subtlety worth keeping in mind: with lists and strings, slicing always returns a sublist and substring, respectively. Indexing, on the other hand, returns an element of the list or string.

3.4.3 Operators for lists vs strings

Lists and strings are examples of sequences with many capabilities beyond accessing elements and sub-sequences. Figure 3.7 highlights the most common of these capabilities: `len` returns the length of a sequence; `in` is a boolean operator that checks for membership at the top-level of a sequence. In addition, the operators `+` and `*` have special meaning for Python sequences. The addition operator *concatenates* two sequences of the same type. That is, adding two sequences results in a new sequence containing all of the elements, in order, of the left- and right-hand operands:

Slicing what to do if you want a bigger piece of the pie!



s = 'harvey mudd college'

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18

s[:] slices the string, returning a substring

the first index is the first character of the slice the second index is **ONE AFTER** the last character

s[0:6] returns 'harvey'

s[12:18] returns 'colleg'

s[17:] returns 'ge'

s[:] returns 'harvey mudd college'

a missing index means the end of the string

Figure 3.5: String-slicing examples

Lists ~ Strings of *anything*

Commas
separate
elements.

L = [3.14, [2,40], 'third', 42]

Square brackets tell python you want a list.

L[0] → 3.14

L[0:1] → [3.14]

L[2][1:3] → 'hi'

L[-1::-2] → [42, [2,40]]

Indexing: could return a different type
Slicing: always returns the same type

Figure 3.6: List-slicing examples

Figure with all of the sequence examples---we have a slide with them all.

Figure 3.7: caption here

```
>>> [ 2, 4 ] + [ 6, 8 ]
[2, 4, 6, 8]
```

Multiplication of a sequence by an integer `n` yields a concatenation of `n` copies of the original sequence:

```
>>> 3 * 'ow! '
ow! ow! ow!
```

Multiplication of one sequence by another is not supported, however!

3.4.4 Converting between data types

Python has a built-in function, `type`, that indicates the type of its input. Here are two examples:

```
>>> type(5)
<type 'int'>

>>> type([])
<type 'str'>
```

As a thought experiment, you might guess at the type of the return value of `type(42)`. You can check your guess with `type(type(42))`

Type names, such as `int` and `str`, are themselves functions that convert data into the named type, when such a conversion is possible. One common conversion, `int`, converts a floating-point value to an integer by truncating that value, e.g., `int(2.9) == 2`. Sometimes conversion results can be surprising—for instance, guess the number of elements in `list("[1,2]")`. When the conversion isn't valid, such as `int("three")`, Python will stop and emit a `ValueError`.

Python can't read!

These conversions are made *implicitly* when computing with different numeric types. For example, the division expression `355.0/113` has a left-hand operand of type `float` and a right-hand operand of type `int`. Python converts the `int` 113 to the `float` 113.0 implicitly before performing the division and yielding the value `3.14159...`

Easy as pi!

Watch out! If both operands are `int`, no conversion is made and the expression `355/113` yields the integer value 3.

Aside: maybe this section should go elsewhere?

3.5 Matters of import

But we've made so little progress with our alien's protein-folding code! Indeed, we've looked only at the first line of the overall program:

```
# a list of initial angles in degrees between protein parts
initial_angles = [-11.37, 8.51, 15.05, -3.96, -26.87, -25.85,
                 25.89]

# set up the graphics
```

```
graphics.createChain(initial_angles)

# run the folding
fold(initial_angles)

# when fold returns, we're finished!
print "I fold, therefore I am."
```

The second line, `graphics.createChain(initial_angles)`, illustrates one of Python's important capabilities. The `createChain` function does not appear in the protein-folding file at all. Rather, it is in the file `graphics.py`. To use code from `graphics.py`, make sure that the `graphics.py` file is in the primary program's directory. Then, at the top of the primary program, include the line

It's coincidence that directories are also called "fold"ers.

```
import graphics
```

This statement says "OK, I will go and grab all of the code from `graphics.py` and allow its use from here on." In this case, the imported code sets up the graphical subsystem, computes the coordinates of the input chain's links, and displays it in a window.

Such an imported file is a *module*, and to call one of its functions, preface the function name with the module name:

```
graphics.createChain(initial_angles)
```

Many useful modules come with Python, and they can be invoked from the command line as well as within a file:

```
>>> import math
>>> math.sqrt(42)
6.4807...

>>> import random
>>> random.choice(["rock", "paper", "scissors"])
"rock"
>>> random.choice(["rock", "paper", "scissors"])
"scissors"
>>> random.choice(["rock", "paper", "scissors"])
"rock"
```

which results in a mean RPS player!

The collection of these *importable* modules is called Python's *library*. It is enormous, hugely useful, and beautifully documented on the Web at <http://docs.python.org/library/>. What's more, most of that documentation is built-in to the language. The `dir` command provides a listing of available functions, and the `help` function provides documentation on each:

```
>>> import math
>>> dir(math)
['__doc__', '__file__', '__name__', '__package__', 'acos', 'acosh',
'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'copysign',
'cos', 'cosh', 'degrees', 'e', 'exp', 'fabs', 'factorial', 'floor',
'fmod', 'frexp', 'fsum', 'hypot', 'isinf', 'isnan', 'ldexp', 'log',
'log10', 'log1p', 'modf', 'pi', 'pow', 'radians', 'sin', 'sinh',
'sqrt', 'tan', 'tanh', 'trunc']
```

```
>>> help(math.radians)
Help on built-in function radians in module math:
```

```
radians(...)
    radians(x) -> converts angle x from degrees to radians
```

```
>>> math.radians(180)
3.1415926535897931
```

Data is available, too, in some modules:

```
>>> import math
>>> math.pi
3.1415926535897931
```

```
>>> math.e
2.7182818284590451
```

Aside: should we mention the `from graphics import *` option?

3.6 Functions are data, too: reduce, map, and lambda

3.6.1 Motivating reduce

From our detour through Python's library we return to the "fold":

```
def fold(old_angles):
    """fold
        input: a list of angles among the elements in
               the "chain"
        output: None, but each call to fold pushes the
               chain into a lower-energy state (until a
               minimum is reached)
    """
    ALL_angles = all_neighbors(old_angles)

    new_angles = reduce(smaller_E, ALL_angles, old_angles)
```

```
if new_angles != old_angles:
    graphics.updateChain(new_angles) # update graphics
    time.sleep(0.05)    # pause for 0.05 seconds
    fold(new_angles)    # and keep going...

return
```

The first line after `fold`'s docstring computes and returns all of the chains neighboring the input chain, `old_angles` by exactly one degree in each angle. Before diving into `all_neighbors`, let's run it a couple of times to build intuition:

```
>>> angles = [42]
>>> all_neighbors(angles)
[[41], [43]]

>>> angles = []
>>> all_neighbors(angles)
[[]]

>>> angles = [42, 100]
>>> ALL_angles = all_neighbors(angles)
>>> ALL_angles
[[41, 99], [41, 101], [43, 99], [43, 101]]
```

In each case, `all_neighbors` returns a *list of lists*. The base case yields a list-of-an-empty-list; recursive cases return a list-of-lists in which each sublist contains angles that neighbor the input's by 1 degree.

This is involved enough to warrant its own function, `all_neighbors`, the topic of the next sections. For now, suppose that `ALL_angles` we have the return value from `all_neighbors`. Specifically, imagine the final example above, in which

```
# after the call ALL_angles = all_neighbors(angles)
>>> ALL_angles == [[41, 99], [41, 101], [43, 99], [43, 101]]
True
```

The next line, `new_angles = reduce(smaller.E, ALL_angles, old_angles)`, computes which of the four elements of `ALL_angles` has the smallest internal energy—that smallest-energy list will be the `new_angles` as the simulation continues.

3.6.2 `reduce`: a function that takes a function as input

The built-in Python function `reduce` accomplishes this task. `Reduce` takes three inputs: the second input *must be* a list that you'd like “reduced” to a single element of that list. The third input is a “base case” of sorts. More detail to come. . . .

But it's the first input to `reduce` that makes `reduce` different from the previous functions we've seen. That first input to `reduce` must itself be a function. It's said that in Python functions are first-class data, because they themselves can be used in the same way that data like numbers, strings, and lists. Functions can be inputs to

functions, the outputs of functions, and can even remain nameless, all in the way other data can.

So, `reduce`'s first input must be a function. That input function must, in turn, take two inputs—two inputs *of the type contained in the list that is the second input*. Then, `reduce` applies its first input's function “through” the second input's list.

Apply a function “through” a list? Let's explain this with some examples. We'll use `min` and `max`, functions built-in to Python that can take two inputs: `min(42,5)` returns 5 and `max(42,5)` returns 42.

Then,

```
# An example of reduce using min:
>>> reduce(min, [5,42,3,101010,2,18])
2
```

Here, `reduce` first applies `min` to the initial two elements of the list. When it gets the result of `min(5,42) == 5`, it applies `min` to *that result* of 5 and the next element in the list: `min(5,3)`. It gets the result, 3, and applies `min` to 3 and 101010. This process continues, making the calls `min(3,2)` and then `min(2,18)`, yielding a final return value of 2.

Thus, `reduce` has use the two-input function `min` to find the overall minimum of an arbitrary list of elements.

Similarly,

```
# An example of reduce using max:
>>> reduce(max, [5,42,3,101010,2,18])
101010
```

3.6.3 Reducing nothing at all?

What if we call `reduce` with an empty list? Python has no data to work with, so it reports an error:

```
# An example of reduce using min:
>>> reduce(min, [])

Traceback (most recent call last):
  File "<pyshell#32>", line 1, in <module>
    reduce(min, [])
TypeError: reduce() of empty sequence with no initial value
```

Because of this, `reduce` allows an optional third input. That third input is the base-case value returned when the input list is empty. Also, it is the first value used when the input list isn't empty:

```
# An example of reduce using min:
>>> reduce(min, [5,42,3,101010,2,18], -1)
-1
```

because the `-1` is smaller than everything else in the list. In short,

```
reduce(f, L, e) == reduce(f, [e]+L)
```

3.6.4 One chain to rule them all

The power to `reduce` extends to *any* two-input function, including our own:

```
def plus(x, y):
    return x + y

>>> reduce(plus, [3,4,5])
12

>>> reduce(plus, [1,3,5,9,11,13])
42
```

This will be the approach we use to find the minimum-energy molecule in our folding simulation.

Because we want to reduce our list of possible chains to a single, minimum-energy chain, `reduce` is the right approach. We need our own two-input function `smaller_E` that returns which of two inputs configurations requires less energy to maintain. Our `smaller_E` does only that; it delegates the actual energy computation to another call:

```
def smaller_E(angles1, angles2):
    """smaller_E
        two inputs: both are lists of bond angles within a molecule
        one output: the list of bond angles with smaller energy (E)
    """
    if compute_E(angles1) < compute_E(angles2):
        return angles1
    else:
        return angles2
```

Thus, the `fold` function makes the call

```
new_angles = reduce(smaller_E, ALL_angles, old_angles)
```

which finds the smallest-energy chain among `ALL_angles` and `old_angles` and then names the result `new_angles`.

3.6.5 A map of map

What about that energy computation itself? Writing `compute_E` only postpones the inevitable. It is in `compute_E` where CS and BioChem will meet. Here's the plan:

- Find all of the 2d coordinates of the vertices of the chain.
- Find all of the *pairwise distances* between pairs of those coordinates.
- Compute the energy for each of those pairs
- Sum up those pairwise energies.

In Python, this becomes `compute_E`:

```
def compute_E(angles):
    """compute_E
       input: a list of bond angles
       output: the energy (E) required to maintain
              those angles
    """
    points = graphics.compute_coordinates(angles)
    pairwise_distances = all_pairwise_distances(points)
    E = sum(map(pairwise_energy, pairwise_distances))
    return E
```

What is this `map`? Like `reduce`, `map` takes a function `f` as its first input and a list `L` as its second input. Then, `map` returns a *new list* in which `f` has been applied to each element of `L`. Here are three examples using the built-in `abs`, `len`, and `sum`:

```
>>> map(abs, [-5,-42,20,-1])
[5, 42, 20, 1]

>>> map(len, [[], [42], [42,42], [42,42,42]])
[0, 1, 2, 3]

>>> map(sum, [[], [42], [42,42], [42,42,42]])
[0, 42, 84, 126]
```

In `compute_E`, above, the call `map(pairwise_energy, pairwise_distances)` returns a list of the energy required to maintain each of the distances in `pairwise_distances`. The call to `sum` returns the total energy, which is named `E`.

3.6.6 Use it or lose it

The `all_pairwise_distances` function takes in a list of coordinate pairs (“points”) and returns all of the Euclidean distances among pairs of those points:

```
>>> points = [[0,0], [3,4]]
>>> all_pairwise_distances(points) # only one distance
                                   # between two points
[5.0]

>>> points = [[0,0], [3,4], [8,-8]]
>>> all_pairwise_distances(points) # three distances
                                   # among three points
[5.0, 11.313708498984761, 13.0]
```

and so on.

Writing `all_pairwise_distances` follows an algorithm-design strategy nicknamed “use it or lose it.” The idea is to single out the first point in the input list, and then split the problem into two parts:

- Finding all of the pairwise distances that **use it**—all the distances from other points to the singled-out point
- Finding all of the pairwise distances that **lose it**—all the distances that don't use the singled-out point at all

The “lose it” half is easier, it's simply a recursive call using the rest of the list! The “use it” half requires finding the distance from each of the rest of the points to the first, singled-out point—a perfect opportunity to use `map`! Here is `all_pairwise_distances`:

```
def all_pairwise_distances(points):
    """all_pairwise_distances
       input: a list of lists of two elements (points)
       output: a list of all pairwise distances among
              the input points
    """
    if len(points) < 2: return []

    first = points[0]
    rest = points[1:]

    dists_without_first = all_pairwise_distances(rest)
    dists_with_first = map(dist_from(first), rest)

    return dists_with_first + dists_without_first
```

There is one surprise here: the call `map(dist_from(first), rest)`. Remember that `map` requires its first argument to be a function. Thus, `dist_from(first)` must return a *function*! The next section shows how this works in Python.

3.6.7 Functions returning functions

The `dist_from(point)` function takes as input a coordinate pair—a list of two elements. In order to be used by `map`, above, it needs to return a function. In particular, it needs to return a function that returns the distance of its input to `dist_from`'s input, `point`!

Python makes it easy to define—and return—a function from within a function:

```
def dist_from(point):
    """dist_from
       one input: a list of [x,y] coordinates (a point)
       one output: a _function_ that computes distances
                  to the input point!
    """
    x1, y1 = point # "unpack" point into its two coordinates

    def f(point2):
        x2, y2 = point2
```

```
    dist = math.hypot(x1-x2, y1-y2) # built in to the
                                    # math module
    return dist

    return f
```

That's all there is to it! Plus, we've leveraged `math.hypot`, the hypotenuse-finding function in the `math` module.

3.6.8 The biochemistry, at last!

It is `compute_E`'s function `pairwise_energy` that models the energy required to maintain a particular distance between two links in our protein chain:

```
def pairwise_energy(dist):
    """pairwise_energy
        This is where the bio/chem appears (or should)
        input: a single value, the distance between two pieces
              (peptides) of the molecule
        output: the energy required to maintain that distance
    """
    # "Steric" energy is required to hold atoms or molecules
    # so close that their volumes overlap. We use an exponential:
    steric_energy = math.exp((2*20)/dist)
    # "Van der Waals" forces pull molecules to a "preferred" mutual
    # distance of 60. Here, quadratic. More formal estimates are **6 !
    van_der_waals_energy = abs(dist-60)**2
    # we return the sum of these two energies
    return steric_energy + van_der_waals_energy
```

This is a remarkably rough approximation, but because it has been modularized into its own well-defined function, it becomes easy to experiment with. For instance, the alternative

```
def pairwise_energy(dist):
    """pairwise_energy, integrated from gravity's inverse-square law
    """
    G = 6.673e-11 # Gravitational constant, G
    return -G/dist
```

yields the gravitational energy between two unit-mass objects, as it depends on the distance `dist` between them.

More accurate estimates of the chemical pairwise energy between the links in protein chains would require distinguishing among the 20 different link (amino-acid) types, resulting in 390 possible pairs of types. In addition, each link in a real protein has two angles, not one, and is represented by 3d coordinates, not 2d.

The alien promises a full accounting of these nuances in the soon-to-be-released version 2.0 of *SuperFunFold*.

3.6.9 Anonymous functions: lambda

Only one function from our original top-down design of *SuperFunFold* remains: `all_neighbors`. On input of a single list of angles, `all_neighbors` seeks to return a list of lists of angles, each a neighbor of the input, in the sense that its angles are one degree away from each of the input's corresponding angles. For example,

```
>>> all_neighbors([5, 42])
[[4, 41], [4, 43], [6, 41], [6, 43]]
```

It seems `all_neighbors` lends itself to a recursive solution:

- If the input list is empty, it has no neighbors, so the result is `[[]]`.
- Split the input list into a first angle `a` and a list of the rest of the angles.
- Find all of the neighbors of the list-of-the-rest-of-the-angles. Call the result `N`.
- Add the angle `a+1` to each element of `N`—this is one half of the full set of neighbors.
- Add the angle `a-1` to each element of `N`—this is the other half of the full set of neighbors.

The Python code follows this top-down design, now at the level of a single function, closely. The one challenge appears in the steps that “add an angle to each element of `N`”. Since it’s doing something to each element of a list, it’s a great opportunity to use `map`! But how could we write a function that *prepends* a particular angle to each element? Here is the code with placeholders in those spots:

```
def all_neighbors(angles):
    """all_neighbors
       input: a list of angles
       output: a list of lists of angles, with each angle 1
              degree away from its corresponding input angle
    """
    if angles == []: return [[]]

    a = angles[0]
    rest_of_angles = all_neighbors(angles[1:])

    return map(FUNCTION_THAT_PREPENDS_[a-1], rest_of_angles) + \
           map(FUNCTION_THAT_PREPENDS_[a+1], rest_of_angles)
```

We *could* write one or more helper functions that return an appropriate function for each of those two cases, as we did with `dist_from`.

But Python offers a quicker alternative when you need a small, task-specific function: *anonymous functions*. An anonymous function defines a small function without the syntactic overhead of `def`, indented blocks, and `return`. The keyword used to introduce an anonymous function is `lambda`; for this reason anonymous functions are sometimes called `lambda`-expressions.

The expression `lambda x: x + 1` is an example of an anonymous function with a single input `x` and whose output is `x+1`. This `lambda`-expression can be used anywhere that a function could be, such as in a direct application:

```
>>> (lambda x: x + 1)(41)
42
```

But anonymous functions are most common as an input to `map`:

```
>>> map(lambda x: x + 1, [1,2,3])
[2,3,4]
```

or, a two-input anonymous function for `reduce`:

```
>>> reduce(lambda x,y: x*y, [1,2,3,4])
24
```

The anonymous functions needed by `all_neighbors` are short:

```
return map(lambda R: [a-1]+R, rest_of_angles) + \
        map(lambda R: [a+1]+R, rest_of_angles)
```

and sweet!

3.6.10 And beyond...

Takeaway message: *This chapter's takeaway message is that functions can be used in all of the same ways that data can be: as inputs, outputs, and without explicit names.*

This *functional programming* approach offers specific tools—`map`, `reduce`, and `lambda`—and general design strategies, top-down and “use it or lose it” approaches, to help you built software that is simultaneously powerful, succinct, and flexible.

Happy functioning!

Chapter 4

Computer Organization

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

4.1 Introduction to Computer Organization

When we think of a computer, we tend to focus on its most visible aspects: the keyboard, display, and mouse. But what goes on inside the box? How does the computer do all those fancy things? Recursion, for example, seems like magic—is the computer *actually* a magical device?

In this chapter, we'll be looking at *computer organization*: the techniques used to turn rivers of electrons into useful computation. We'll work from the ground up, starting with a look at how to represent information electronically, moving on to computational circuits, using those circuits to build a full-blown computer and, finally, program that computer in its own native language. Ultimately, we'll see how programs, including recursive ones, really work. The “magic” will be demystified!

A modern Intel Pentium® chip can contain nearly a *billion* transistors, each of which has a specific job to do. But don't worry—we won't be getting quite that complicated!

All right, a “partially-blown” 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

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

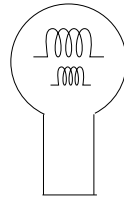


Figure 4.1: A three-way light bulb.

That is a shocking idea!

In other words, the current way of doing things is not revolting.

A three-way lamp really has *four* switch positions: off, and three increasing brightness levels.

Watt have light bulbs got to do with computing!?

A bright idea!

Sorry, we're using the margin! Go find a different place to try this.

several problems with this approach, not the least of which would be the need to have a million-volt computer! Instead, we use a simpler system: everything is built up using sequences of zeros and ones. This allows us to use lower voltages and simpler circuitry.

It is easy to see how, for example, a light could be used to represent the numbers 0 and 1. We can agree ahead of time that when the light is off, it means 0, and when it's on, we are representing 1. What about bigger numbers? Well, light switches can do that as well.

Consider the common “three-way” lamp. 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 one, the other, or both, we can get 50, 100, or 150 watts' worth of light.

This idea can be carried further; for example, a bulb with filaments of 50, 100, and 200 watts could produce any of 0, 50, 100, 150, 200, 250, 300, or 350 watts by choosing various combinations. These particular seven wattages aren't all that useful to us, so we might instead use these values to represent the seven integers 0 through 6. That is, we could agree that 0 watts represents the number 0, 50 watts represents the number 1, and so forth.

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 2^0 watt filament, a 2^1 watt filament, and a 2^2 watt filament. Then we could make 0 watts by turning none of the filaments on, we could make 1 watt by turning on only the 2^0 watt filament, we could make 2 watts by turning on the 2^1 watt filament, and so forth up to $2^0 + 2^1 + 2^2 = 7$ watts using all three filaments.

Imagine now that we had different powers of 2 to use in constructing our number: $2^0, 2^1, 2^2, 2^3$. In the margin of this book, take a moment to try to write the numbers 0, 1, 2, and so forth as high as you can go using 0 or 1 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:

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

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 **binary digits**. *Binary* means “using two values”—the 0 and the 1 are the two values here.

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* everyday. 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 lived in Northern California and 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.

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 $b^d - 1$ 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 computers use 32 bits to represent a number in base 2. This is called a “32-bit computer”. Therefore, we can uniquely represent all of the positive integers between 0 and $2^{32} - 1$. That’s 4,294,967,295. A 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

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 leftmost position and carry the 1. That 1 represents a 10 and is therefore propagated to the next column representing the “10’s place”.

Adding 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 leftmost (or “least significant”) column and add. That sum is 1. Now we move to the next column. This is the 2^1 position or “2’s place”. Each of our two numbers has a 1 in this position. $1 + 1 = 2$

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

That’s a bit of an odd contraction!

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

In case you’re wondering, that’s BIG!

Sum fun with addition.

but we only use 0's and 1's in base 2. This is analogous to adding $7 + 3$ in base 10. Rather than writing a 10 in that case, 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 $2 \cdot 2^i = 2^{i+1}$.

We'll try not to get carried away with these examples, but you should try adding a few numbers in base 2 to make sure that it makes sense to you.

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

4.2.3 Negative Numbers

Now that we are masters of arithmetic in base 2 (and thus in all bases!), let's revisit numbers briefly. In particular, while we know how to represent positive integers, how about representing negative numbers. One approach might be to reserve one bit to represent the sign of a number; for example, in a 32-bit computer we might use the leftmost bit for this purpose: Setting that bit to 0 the number 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*.

It's so clever that it deserves two compliments.

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 a 32-bit or a 64-bit computer), 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 10000000; 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.

4.2.4 Fractional Numbers

Integers are useful for a lot of things, but sometimes we need to work with numbers fractional parts. For example, it would not be easy to get by without π !

Proving that the expression “as easy as pi” is a misnomer.

One approach (which is often used in video and music players) is to just agree among ourselves 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, 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 8 digits to represent our numbers. We might use the first 6 digits to represent a number, with the convention that there is an implicit 0 followed by a decimal point right before the first digit. For example, the 6 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, 123345678 would represent $0.123456 \cdot 10^{78}$.

Computers use a similar idea to represent fractional numbers except that base 2 is used in base 10. Moreover, we allow for negative exponents in order to represent very small numbers. We also allow for the entire number to be negative. An important detail that is often forgotten (but which we will see again later in this book) is that we cannot represent all real numbers precisely, not even those in the range from 0 to 1. The reason is that with a finite number of bits, we can only represent a finite number of different numbers. However, there are an infinite number of real numbers—even just in the range from 0 to 1. If you'd like to learn more about fractional numbers (usually called “floating point”), you can read the Wikipedia article on IEEE floating point, or visit <http://steve.hollasch.net/cgindex/coding/ieeefloat.html>.

Avogadro's number is approximately 6.02×10^{23} .

4.2.5 Letters and Strings

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.

It's fairly easy to represent the alphabet numerically; we just need to come to an agreement 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. But if we want to send a document to a friend, it helps to have agreement. Long ago, the American National Standards Institute (ANSI) published such an agreement, called *ASCII* (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.

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

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

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

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 symbols '9'.

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.) 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 cure that oversight, the International Standards Organization (ISO) eventually devised a system called Unicode, which can represent every character in every 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.

Of course, individual letters aren’t very interesting. Humans normally like to string letters together, and computer scientists do the same. 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 is that when you are given a long string of numbers, you have to know when to stop; the usual 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: for example, 9, 99, 104, 111, 99, 111, 108, 97, 116, 101.

4.2.6 Structured Information

Using the same concepts, we can represent almost any information as a sequence of numbers. For example, a picture is just 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.

In each case, all we need is a convention as to what the numbers represent and how they should be interpreted by the computer. Complex objects like movies can be built up from simpler ones like pictures, which in turn are built from even simpler ones like numbers (which are themselves built from collections of bits). For bulky objects like movies, we normally use compression algorithms to save space.

4 bits are sometimes called a “*nybble*”; we take no responsibility for this pathetic attempt at a pun.

There are even unofficial Unicode symbols for Klingon!

A two-hour movie at TV quality contains nearly 160 billion bytes.

George Boole

George Boole (1815-1864), an English mathematician, gave us the foundation of computation in two pamphlets called *Mathematical Analysis of Logic* and *Laws of Thought*. Using techniques similar to those in standard algebra, he showed how to symbolically manipulate two-valued variables (i.e., 0 and 1, or **true** and **false**) in a consistent manner. Using his methods, it was possible to engage in logical reasoning by following simple mathematical principles.

In the mid-1930's, Claude Shannon (1916-2001) realized that Boole's ideas could be applied to the construction of electrical circuits. Shannon recognized that Boole's 0 and 1 could be represented as a switch being "on" or "off," and that his mathematics could be applied to designing circuits that could carry out useful computations.

4.3 Logic Circuitry

Now that we have adopted some conventions on the representation of data it's time to build devices that manipulate this data. In other words, it's time to start building a computer! We'll begin with the logic of manipulating 0's and 1's, called *Boolean algebra*. From there, we'll build circuits that implement this logic and from there we'll build the elements of a real computer.

4.3.1 AND, OR, and NOT

We've actually seen the basics of Boolean algebra in Chapter 2 when we talked about Python's **and**, **or**, and **not** statements. The essence of Boolean algebra is entirely captured in these three simple operations. In this chapter, we'll use the upper-case AND, OR, and NOT to indicate that we are talking about operations on bits rather than Python. Recall that *AND* is a function that operates on two variables, each of which can be 1 or 0 (alternatively, **true** or **false**). We define x AND y to be 1 if and only if both x and y are 1, and 0 otherwise.

One way of specifying a Boolean function is with a *truth table*, which is simply 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 true if and only if both of its arguments are true. If either argument is false, AND is false. If x is 0, the result of AND is 0. If x is 1, the result of AND is y . The same is true for y .*

We are writing these in upper-case letters because we are going to capitalize on the power of AND, OR, and NOT!

OR is a two-argument function that is true 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; the first three lines of the above table are indeed identical to addition, but note that the fourth is different.

Takeaway message: *OR is true if either of its arguments is true. It is false if and only if both arguments are false. If x is 0, the result of OR is y . If x is 1, the result of OR is 1. The same is true for y .*

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. In this section we'll show how to do that. This fundamental result will allow us to build circuits 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.

Let's 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 only outputs 1 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.

Perhaps “minterms” should be called “happyterms”.

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 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 second, third, and fourth rows of the truth table as it should. How about the first row, the “uninteresting” case where $x = 0, y = 0$ 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 $x = 1, y = 0$ “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 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. Now AND all of these variables together.
4. Combine all the 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 NOTs and ANDs

Figure 4.2: An electromagnetic switch.

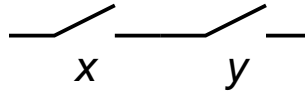


Figure 4.3: An AND gate constructed with switches .

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 useful because it allows us to construct a logical expression for any truth table. Notice, however, that it 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.

As we have seen, the minterm expansion principle doesn’t always produce the simplest possible expression. However, it is guaranteed to always produce a correct result, whereas inspecting the truth table and trying to write an expression is hard and often leads to errors. For that reason, the minterm expansion is a good approach for creating Boolean expressions. Moreover, the minterm expansion principle is an algorithm, so it can be implemented on a computer and performed automatically!

4.3.3 Logic Using Electrical Circuits

Claude Shannon’s great insight was that it is possible to implement AND, OR, and NOT using electrical circuits. Let’s imagine that our basic “building block” is an electromagnetic switch as shown in Figure 4.3.3. When the electrical input x is “low”, the switch stays open and thus no electrical signal flows from the power source to the output. When the input on x is “high”, the magnet is activated and the switch closes, allowing an electrical signal to flow from the power source to the output.

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. Notice that when either x or y or both 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

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! Very nice because computers perform laborious tasks happily and quickly.

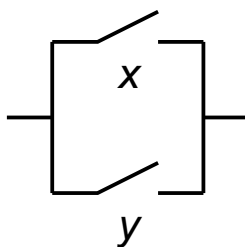


Figure 4.4: An OR gate constructed with switches.

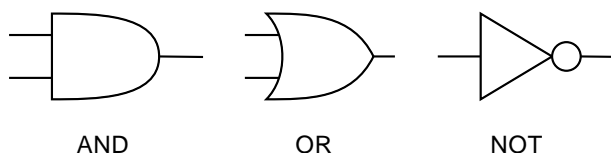


Figure 4.5: Symbols used for AND, OR, and NOT gates.

function NOT x can be implemented by constructing a switch that conducts if and only x is 0.

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 considerably smaller, more reliable, and more efficient. Since the details of the switches aren't terribly important to us, we abstract the gates using symbols as shown in Figure 4.3.3.

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.

The symbols for AND and OR are weird, and it's easy to confuse them.

4.3.4 Computing With Logic

Now that we know how to implement functions of Boolean variables, let's look at how we can 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:

Figure 4.6: A circuit for the implication function.

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 rightmost 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 might have to carry to the next column) we add the digits in the next column and so forth, proceeding from one column to the next, until we're done. 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. Moreover, this "modular" approach is elegant. It allows us to 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). This idea is idea of modularity and reuse is central in computer science.

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 s) 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 s , use the minterm expansion principle to write a logic expression for s . 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—its 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

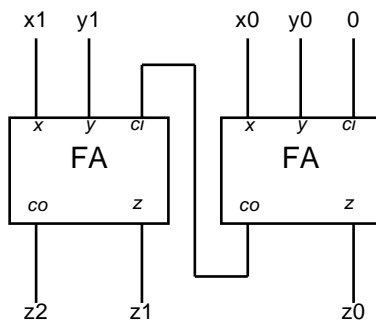


Figure 4.7: A 2-bit ripple-carry adder. Each box labeled “FA” is a full adder that accepts two input bits plus a carry, and produces a single output bit along with a carry into the next FA.

many gates would be used in total for a 16-bit ripple-carry adder? Your answer will be in the hundreds rather than the billions required in our first approach!

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 could do so from scratch using the minterm expansion principle or we could observe that multiplication involves a number of addition steps. Using the latter approach, we could use our modular design approach to build a multiplier out of multiple ripple-carry adders. In any case, using just the minterm expansion principle and modular design, we can now build virtually 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 use the principle of modular design to assemble many latches into a device that stores a lot of data.

We almost didn’t remember to talk about this!

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	$x \text{ NOR } y$
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).

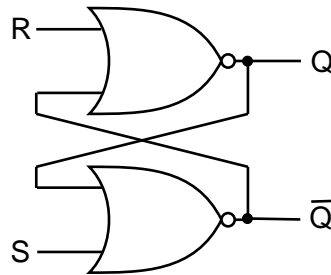


Figure 4.8: A latch built from two NOR gates.

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.

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 out this though 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 1. 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, assert S for a moment; if you want to remember a 0, assert R . 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.

We’re sheepish about making any RAM puns here since 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. 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 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.

A modern computer operates on the same principle. Your calculator and the scratch sheet correspond to the *central processing unit (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.

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.

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

"Balancing" the club's budget?! OK, OK, wheel make no more unicycle jokes!

See! We "spoke" the truth; no unicycle jokes.

We're sticking to our promise not to "tire" you with unicycle jokes.

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!

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 1.1.1), 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.

add some numbers and store them some place. Then, you fetch the next instruction from the notebook.

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.

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.

One of von Neumann’s colleagues was Dr. Claude Shannon—we mentioned him earlier in this chapter in our aside about George Boole. Shannon, it turns out, was a very good unicyclist.

4.4.1 The von Neumann Architecture

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 2, we could adopt the convention that we'll write "add 2, 0, 2". The first two numbers are the registers where we'll get our input and the third number is the register where we'll store our result. In binary, "add 2, 0, 2" would be represented as "00 10 00 10". We've added the spaces to help you see the numbers 00 (indicating "add"), 10 (indicating register 2 as the first register that we will add), 00 (indicating register 0 as the second register that we will add), and 10 (indicating register 2 as the destination where we will store the result).

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 (S0 and S1) will encode the register containing the first input to our instruction, the next two bits (T0 and T1) will encode the register containing the second input to our instruction, and the last two bits (D0 and D1) will encode the destination register where we will save the result of our instruction. This representation is shown below.

I0 I1	S0 S1	T0 T1	D0 D1
-------	-------	-------	-------

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.

The computer begins by going to this memory location and fetching the data that resides there. Now look at the memory, shown on the left 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 its an addition instruction. the following 10 and 00 indicate that we need to get the data stored in registers 2 and 0, respectively. These values are then sent to the CPU's ripple-carry adder where they are added. The final 10 in the instruction register indicate that the result of the addition should be stored in register 2. 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.

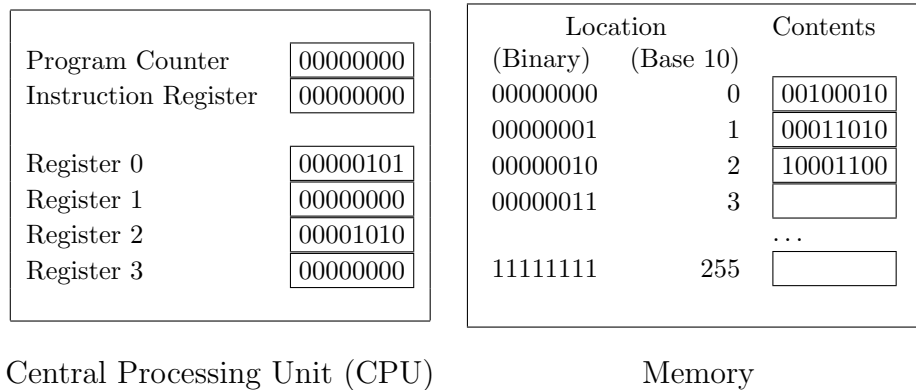


Figure 4.9: A computer with instructions stored in memory. The program counter tells the calculator where to get the next instruction.

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 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.)

“Wait!” we here 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:

- Move information between the registers and a large memory,
- Get data from the outside world,
- Print results, and
- 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:

- While our simple computer used 8-bit instructions, in HMMM, both instructions and data are 16 bits wide. That allows us to represent a reasonable range of numbers, and lets the instructions be more complicated.
- 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.
- 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.)

Since the HMMM instruction format has quite a few instructions, it's inconvenient to try 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
```

A very simple process is used to convert this mnemonic assembly language into the 0's and 1's—the “machine language”—that the computer actually understands.

A complete list of HMMM instructions, including their binary encoding, is given in Figure 4.10 on page 93.

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 illustrate several important features of HMMM. (We suggest that you follow along by downloading HMMM from <http://www.cs.hmc.edu/hmmm> 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:

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.

HMMM's instruction set is large, but not hmmmungous.

Is there a good mnemonic for remembering how to spell “mnemonic”?

Not the least of which is that writing even a short program in assembly language can be a hmmmbling experience!

```
#
# 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      loadn  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 on page 81. 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 `MULT` 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. The 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, `loadn` (*load number*), lets us insert a small constant into a register. (“Load” usually means “grab something and put it in a register”; the converse is to “store” the contents of a register into memory.) As with `mul`, the destination is given first; this is equivalent to the Python statement `R2 = 2`.

The `DIVide` 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, it will crash. So we need to tell it to `halt` after it’s done with its work.

That’s it! But will our program work correctly?

It would be hmmmurous if *we* got this wrong.

Trying It Out

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 r1
3 : 0001 0010 0000 0010      3      loadn  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
```

¹On 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.

```
Enter number: 5
12
```

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 it 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      loadn  r2 2
4      div    r1 r1 r2 # Divide by 2
5      write  r1
6      jump   0
```

then our program will calculate triangle areas forever.

What's that `jump 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”):

```
% ./hmmmSimulator.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 `jump` 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 `jeqz` which is pronounced "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.

```

#
# 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      jeqz   r1 9     # Jump to halt if base is zero
2      read   r2      # Get height
3      jeqz   r2 9     # Jump to halt if height is zero
4      mul    r1 r1 r2 # b times h into r1
5      loadn  r2 2
6      div    r1 r1 r2 # Divide by 2
7      write  r1
8      jump   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   loadn  r2,1
2   jeqz   r1,6     # Quit if N has reached zero
3   mul    r2,r1,r2 # Update product
4   addn   r1,-1    # Decrement N
5   jump   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 `loadn` 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 $\binom{n}{k} = \frac{n!}{k!(n-k)!}$. 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.

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      mov    r1,r3   # Calculate N!
3      call   r14,15  # ...
4      mov    r5,r2   # Save N! as C(N,K)

5      mov    r1,r4   # Calculate K!
6      call   r14,15  # ...
7      div    r5,r5,r2 # N!/K!

8      sub    r1,r3,r4 # Calculate (N-K)!
9      call   r14,15  # ...
10     div    r5,r5,r2 # C(N,K)

11     write  r5      # Write answer
12     halt

13     nop
14     nop

```

```
# Factorial function.  N is in R1.
# Return address is in R14.
15    loadn  r2,1    # Initial product
16    jeqz   r1,20   # Quit if N has reached zero
17    mul    r2,r1,r2 # Update product
18    addn   r1,-1   # Decrement N
19    jump   16     # Back for more

20    jumpi  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.

One of the hmmm-dingers
of programming in
assembly language!

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 `nops` 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 `call` instruction, which appears at lines 3, 6, and 9. `Call` partly works just like `jump`: it causes HMMM to start executing instructions at a given address, in this case 15. But if we had just used a `jump`, after the factorial function calculated its result, it wouldn't know whether to jump back to line 4, line 7, or line 10! To solve the problem, the `call` uses register R14 to save the address of the instruction immediately *following* the call.²

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 `jumpi` (jump *indirect*) 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), `jumpi` goes to a *variable* address taken from a register. In other words, if R14 is 4, `jumpi` will jump to address 4; if it's 7 it will jump to 7, and so forth.

4.5.4 Recursion

In Chapter 3 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 first need to discuss a very cool technique called a *stack*.

Stacks

You may recall that in Chapter 2 we talked about stacks (remember those stacks of lock 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

²We could have chosen any register except R0 for this purpose, but by convention HMMM programs use R14.

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 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 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. The stack will now look like:

	Address	Contents
R15 →	103	23
	102	42
	101	56
	100	12

Later, to *pop* the top value off, we must ask R15 where the top is⁴ (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
storei r4,r15  # Store r4 on the stack
```

The `storei` (*store indirect*) 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 `storei` can put things onto the stack, `loadi` will recover them:

³We say that R15 *points to* location 102.

⁴We say we *follow the pointer* in R15.

```
loadi 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 `storei`; in the second, it is decremented *after* the `loadi`. 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 an 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.

Many newcomers to HMMM try to take shortcuts with stack saving and restoring. The common mistake 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 an 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      loadn   r15,100 # Initialize stack pointer
1      read    r2      # Get N
2      call    r14,5   # Recursive function finds N!
3      write   r1      # Write result
```

It's a common mistake,
but to err is hmman.

```

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      jeqz    r2,18    # Test for base case (0!)

6      addn    r15,1    # Save precious possessions
7      storei  r2,r15   # ...
8      addn    r15,1    # ...
9      storei  r14,r15  # ...

10     addn    r2,-1    # Calculate N-1
11     call    r14,5    # Call ourselves recursively to get (N-1)!

12     loadi   r14,r15  # Recover precious possessions
13     addn    r15,-1   # ...
14     loadi   r2,r15   # ...
15     addn    r15,-1   # ...

16     mul     r1,r1,r2 # (N-1)! times N
17     jumpi   r14      # Return to caller

# Base case: 0! is always 1
18     loadn   r1,1
19     jumpi   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 to 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 $(N - 1)!$ (10–11) and then recover registers R14 (12) and R2 (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`, `jnez`, `jgtz`, and `jltz`; we trust that those don’t need separate explanations. For convenience, Figure 4.10 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 `call` relate to `jump`? 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 a programming language such as Python? Let’s put Python aside for a moment. In *some* programming languages (e.g. C++) the entirety of the program is first “compiled” into machine language (the binary equivalent of assembly language) using a program called a *compiler*. This compiled program looks like the HMMM code that we’ve written and is executed on the computer. In other languages (e.g. Python and Java), each line of the program is translated one-line-at-a-time into machine language by a program called an *interpreter*.

What’s the difference between the two approaches? 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 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.

Takeaway message: *We’ve now seen how a computer works from the ground up and how to write complex programs on in the language of the machine!*

Note that `sub` can be combined with `jltz` to evaluate expressions like $a < b$.

No more hmmm-ing is music to my ears!

Assembly	Format	Opcode	X	Y	Z	Description
halt	B	0000	0000	0000	0000	Halt program
read rX	B	0000	xxxx	0000	0001	Read user input into register rX
write rX	B	0000	xxxx	0000	0010	Print the contents of register rX on the screen
jumpi rX	B	0000	xxxx	0000	0011	Set program counter to address in rX
loadn rX, n	B	0001	xxxx	nnnn	nnnn	Load the small signed constant n into register rX
load rX, n	B	0010	xxxx	nnnn	nnnn	Load register rX with memory word at address n
store rX, n	B	0011	xxxx	nnnn	nnnn	Store contents of register rX into memory word at address n
loadi rX, rY	A	0100	xxxx	yyyy	0000	Load register rX from memory word addressed by rY: $rX = \text{memory}[rY]$
storei rX, rY	A	0100	xxxx	yyyy	0001	Store contents of register rX into memory word addressed by rY: $\text{memory}[rY] = rX$
addn rX, n	B	0101	xxxx	nnnn	nnnn	Add the small signed constant n to register rX
add rX, rY, rZ	A	0110	xxxx	yyyy	zzzz	Set $rX = rY + rZ$
mov rX, rY	A	0110	xxxx	yyyy	0000	Set $rX = rY$
nop	A	0110	0000	0000	0000	Do nothing
sub rX, rY, rZ	A	0111	xxxx	yyyy	zzzz	Set $rX = rY - rZ$
mul rX, rY, rZ	A	1000	xxxx	yyyy	zzzz	Set $rX = rY * rZ$
div rX, rY, rZ	A	1001	xxxx	yyyy	zzzz	Set $rX = rY / rZ$
mod rX, rY, rZ	A	1010	xxxx	yyyy	zzzz	Set $rX = rY \% rZ$
jump n	B	1011	0000	nnnn	nnnn	Set program counter to address n
call rX, n	B	1011	xxxx	nnnn	nnnn	Set rX to (next) program counter, then set program counter to address n
jeqz rX, n	B	1100	xxxx	nnnn	nnnn	If $rX = 0$ then set program counter to address n
jnez rX, n	B	1101	xxxx	nnnn	nnnn	If $rX \neq 0$ then set program counter to address n
jgtz rX, n	B	1110	xxxx	nnnn	nnnn	If $rX > 0$ then set program counter to address n
jltz rX, n	B	1111	xxxx	nnnn	nnnn	If $rX < 0$ then set program counter to address n

Figure 4.10: The HMMM instruction set.

Chapter 5

Imperative Programming

5.1 Motivation: Measuring the Stroop Effect

Imagine you are a student in an introductory psychology class. You have just learned about the Stroop Effect, a cognitive phenomenon that demonstrates how difficult it is to ignore the semantics of text when attempting to focus on a property of the text itself.

Imagining this is particularly easy if you *are* in an intro psych class right now.

If you're not already familiar with the Stroop Effect, the best way to understand it is to experience it firsthand. Here's what you should do. First, look at the strings of X's in Figure 5.1 and, as fast as you can, state out loud what color each string of X's is. For the most accurate results you should time yourself. Now, look at the words in Figure 5.2. Again, as fast as you can (without making mistakes), say what *color* the text of each word is. Do not read the word itself, rather just say what color its ink is.

You probably found that you could complete the first task much faster than you could complete the second. At the very least, it was probably harder for you to complete the second task because you kept wanting to read the words rather than state their color. If you found that this was indeed the case, then you have just experienced the Stroop Effect—the tendency of the semantic meaning of words to interfere with your cognition regardless of how much you try to ignore it.

However, you might also have noticed that this experiment wasn't very accurate. You probably didn't time yourself that precisely, and it certainly didn't measure up to the standards for controlled experiments. The psychologists who originally discovered the Stroop Effect used paper and stopwatches in their experiments. Now, psychologists rely heavily on computer programs to run their studies.

Figures 5.3–5.6 show an application to measure the Stroop Effect. You can run this application yourself; it is included with the code samples with this book. *Need better instructions about how to run this application and where to get it.* The application shows a series of string using different colors of text; it then asks the user to enter the color of the strings as fast as possible. Some of the strings are simply X's, and others are color names. When the color names *differ* from the color of the text in which they're displayed, our ability to state their color tends to slow down. The application assigns a different key to each color, using keys that the user can easily reach on the



Figure 5.1: Read the color of each line out loud as fast as possible.

BLACK
RED
YELLOW
RED
YELLOW
BLUE
RED
BLACK
BLACK
RED
BLUE
BLACK
BLUE
RED
YELLOW
BLUE
RED
BLACK
YELLOW
RED
BLUE

Figure 5.2: Read the color of each line out loud as fast as possible.

The Stroop Effect

History of the Stroop Effect and some interesting things about it. *How does one cite sources in a sidebar?*

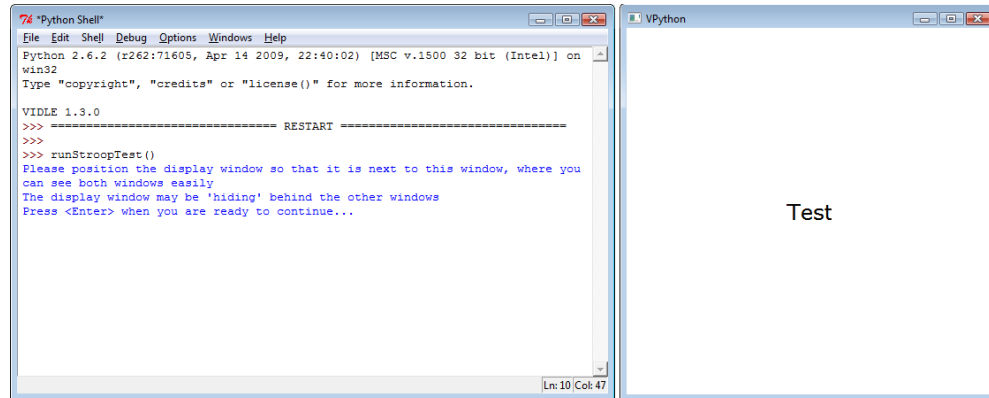


Figure 5.3: The initial screen of an application to measure the Stroop Effect. Colored strings will appear in the window that currently contains the word “Test.”

keyboard. When the user has completed both sets of trials, the program outputs the mean time and standard deviation for each set, e.g.:

```
textData:
    average time:  1.36892001152 seconds;  st dev: 0.581104066849 seconds
nonTextData:
    average time:  0.845719995499 seconds;  st dev: 0.240744437515 seconds
```

At first you might think, “*I could never write this program. That’s way too advanced!*” If these were your thoughts, you’re in for a surprise. In this chapter you will learn all of the tools you need to build this and similar applications.

Before we begin to introduce these conceptual tools, we motivate them by analyzing what the Stroop Effect program needs to do. As we saw in Chapter 3, the essence of any computer program is its data: the form of the input, its transformation (through computation), and the desired output. Take a few moments to think about how you might design 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 essence of the computation remains the same from one implementation to another. The key components that we need to implement this program are:

- The ability to repeat a task (e.g., showing a string to the user) many times.
- The ability to store and manipulate data in different ways (e.g., storing the user’s responses, the time it took the user to respond, and the mapping between colors and keystrokes).

Even if you already know how to write this program, this chapter provides new techniques for doing so simply and elegantly...

5.1. MOTIVATION: MEASURING THE STROOP EFFECT

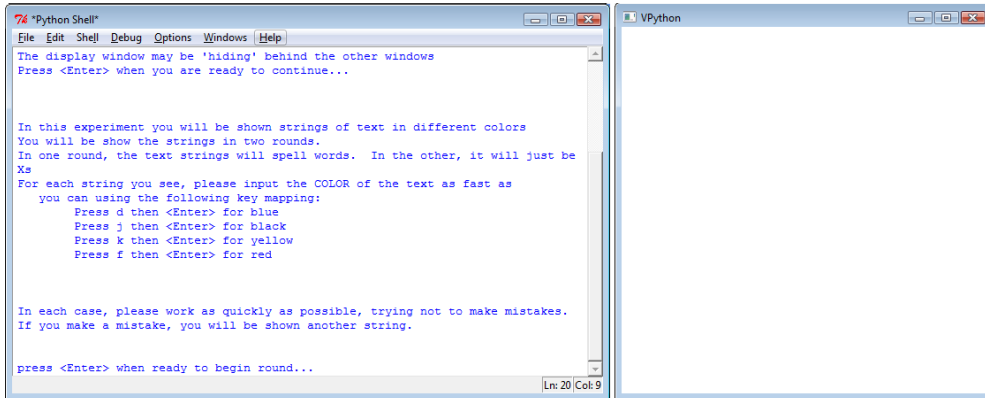


Figure 5.4: The instructions given to the user for the Stroop Test.

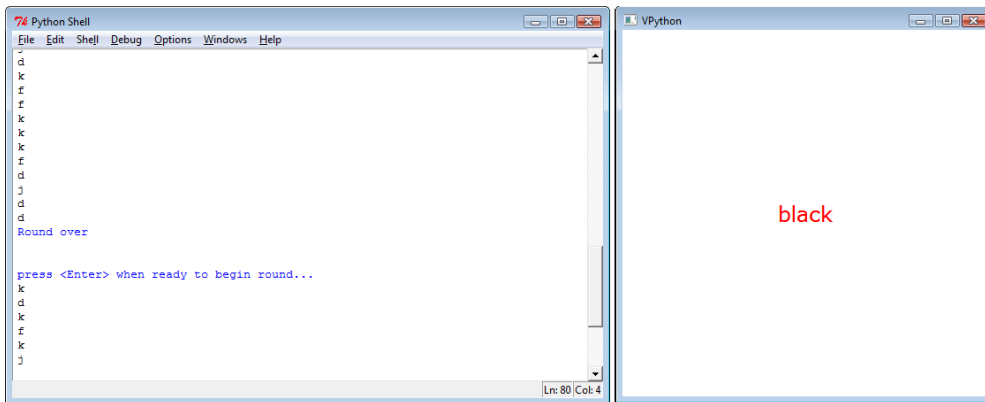


Figure 5.5: A trial with X's.

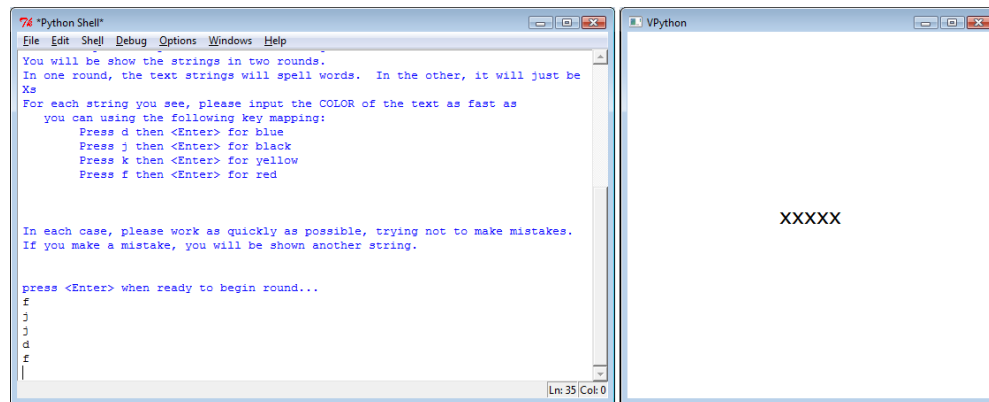


Figure 5.6: A trial with a color name.

- The ability to display colored strings to the user
- The ability to gather input from the user

Throughout the rest of this chapter we will learn ways to perform each of the above tasks.

5.2 Repeated Tasks: Loops

One of the core abilities this program needs to perform a single action (or series of actions) multiple times. We need to show the user a number of example words and then collect and store what color they input. The algorithm that performs this data-collecting part of the experiment looks something like:

- Repeat the following 25 times:
 - Display the name of a color in a different color “ink”
 - Record the user’s answer and the time that it took the user to respond
 - Clear that screen

We could just copy 25 times a piece of code that displays *one* string and collects the user’s response. But the program would be long and inflexible: we may want to change it to 50 trials—or only 10—later on. It would be tedious—and time-consuming—to manually copy and paste the correct part of the code. Furthermore, we might want to let the experimenter set the number of trials on the fly, instead of setting it before the program runs.

We have already seen one way to perform repeated tasks in Chapter 3: recursion. 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. Recursion and loops are equal in their

It’s not worth copying-and-pasting even if you’ve mastered all of your editor’s shortcuts...

computational power: any recursive program can be rewritten using loops (and vice-versa). Some people have a strong preference for one style over the other, but both are fundamental tools for solving computational problems!

5.2.1 Recursion vs. Iteration at the Low Level

In chapters 3 and 4 you saw how to write recursive programs (both in Python and in HMMM). In fact, in Chapter 4 you also saw how to write iterative programs in HMMM, although we didn't call them that. In this section we will define the difference between recursion and iteration using familiar HMMM examples that you have already seen.

To illustrate the difference between iteration and recursion at the HMMM level of abstraction, consider our factorial program. We've already seen a recursive HMMM implementation of factorial:

If you skipped Chapter 4, no worries—you might jumpi to Section 5.2.2.

```

00 read r1          # Read input from the user into r1
01 loadn r15 42     # load the top of the stack (42) into r15
02 call r14 5       # call factorial, return address in r14
03 jump 21         # when factorial returns, skip to the printing
04 nop             # skip a line
05 jnez r1 8        # if n is greater than 0, go to line 8
06 loadn r13 1     # else we're at the base case. Load 1 into r13
07 jumpi r14       # and return it
08 storei r1 r15   # store n on the stack
09 addn r15 1      # increment SP
10 storei r14 r15  # store return address on the stack
11 addn r15 1      # increment SP
12 addn r1 -1      # n = n-1
13 call r14 5      # recursive call to factorial with n-1
14 addn r15 -1     # back from recursive call, decrement SP
15 loadi r14 r15   # restore return address
16 addn r15 -1     # decrement SP
17 loadi r1 r15    # restore n
18 mul r13 r13 r1  # multiply return value from recursive call by n
19 jumpi r14       # return
20 nop             # skip a line
21 write r13       # print factorial(n)
22 halt           # done

```

which corresponds to the following python code:

```

def factorial(n):
    if n == 0:
        return 1
    else
        answer = factorial(n - 1)
        return n * answer

n = input()

```

This figure will be an illustration of the number of n 's on the stack at its deepest value when recursive fact is called with $n=3$.

Figure 5.7:

```
print factorial(n)
```

Certainly, this program does a set of tasks multiple times—namely it calculates the factorial of n , then $n-1$, then $n-2$, etc.—without us having to write a separate set of instructions for each of those values that n takes on.

Now consider the alternative HMMM implementation of factorial that we saw in Chapter 4. The version below is the *iterative* implementation of factorial:

```
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
```

While this version calculates the same factorial function, it does that calculation in a manner very different from the recursive implementation. First, it does not use the stack. In the recursive implementation above, the code repeatedly stores values for n on the stack. As illustrated in Figure 5.7, when the user inputs 3, there are 3 separate values of n (1, 2 and 3) stored on the stack when the code reaches the base case. However, in the iterative implementation, we simply modify n , without storing its value on the stack. In effect, we immediately use the current value of n by multiplying it onto the answer, and then we lose it forever.

Second, the iterative version repeats code differently: the `jump` instruction on line 5 instructs the HMMM interpreter to go back to line 2 and repeat the instruction there, completely forgetting where this jump came from. On the other hand, the “recursive” version jumps to the top of the factorial program, remembering where it came from, so that when the factorial program finishes it can continue where it left off.

These two differences are at the heart of iteration. Iteration does not bother with the stack: it simply modifies the values of its local data to determine when to stop repeating what it is executing. Instead of recursion, iteration relies on loops, small blocks of code that execute (potentially) many times without calling the entire function from the beginning each time.

In the factorial example, lines 2 through 5 form the loop itself— they will repeat over and over until n becomes equal to 0. Note that the *local copy* of n is modified—no value for n is stored on the stack. So once the code changes n 's value, it can never get the old value back.


```

1 data = []
2 for i in [0, 1, 2]:
3     startTime = getTime()
4     showNextWord()
5     answer = getUserAnswer()
6     endTime = getTime()
7     data += [endTime-startTime]
8 print "Congratulations. You're done."

```

Annotations in the code: A red box highlights the line `for i in [0, 1, 2]:` with an arrow pointing to it labeled "loop header". A blue bracket on the left side groups lines 3 through 7, labeled "loop body".

Figure 5.8: A for-loop to collect timing data from the user Fix with `raw_input` and a few more labels...

Now that you have seen iteration at the assembly-language level, we show how Python can do the same thing using `for` and `while` loops.

5.2.2 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 Stroop effect data-collection we motivated above.

To begin, assume the user will respond to only 3 color strings; it will be clear how to increase the number of examples from there. To make the larger structure clear, we abstract away details of the timing and string-display into `getTime` and `showNextWord` functions.

```

1 data = []
2 for i in [0, 1, 2]:
3     startTime = getTime()
4     showNextWord()
5     answer = raw_input()
6     endTime = getTime()
7     data += [endTime-startTime]
8 print "Congratulations. You're done."

```

Figure 5.8 shows an annotated for-loop that collects three pieces of data and accumulates them in the list variable `data`. This for-loop will repeat its body, lines 3–7, exactly 3 times. Let's look closely at how this works. Line 2 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]`.
5. A colon, the end of the header and the start of the for-loop body

Lines 3–7 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 8 above is *not* part of the loop body because it is not indented.

We agree: for-loops look good, but how do they *work*? The for-loop operates by taking the control variable (e.g. `i` above) and sequentially giving it the value of *each* element in the list. Every time the control variable gets a new value, all of the instructions in the loop body are executed, in order. When the bottom of the loop body is reached, execution jumps (just like an unconditional HMMM `jmp`) back to the top of the loop, and the control variable is assigned the next element in sequence. When there are no more values remaining in the sequence, the loop ends and Python continues executing the instructions after the loop body—here, on line 8.

To get a better sense of how this process works, imagine “unrolling” the loop to view, at once, all of the statements that all of the loop iterations will execute. The loop is equivalent to with the following code:

```
data = []

# start loop iteration 1
i = 0
startTime = getTime()
showNextWord()
answer = raw_input()
endTime = getTime()
data += [endTime-startTime]

# go back to the start of the loop for iteration 2
i = 1
startTime = getTime()
showNextWord()
answer = raw_input()
endTime = getTime()
data += [endTime-startTime]

# go back to the start of the loop for iteration 3
i = 2
startTime = getTime()
showNextWord()
answer = raw_input()
endTime = getTime()
```

```

data += [endTime-startTime]

# There are no more values in the list for i, so exit the loop
print "Congratulations.  You're done."

```

So 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 fun!) Instead, we can use the built-in `range` function to automatically generate that list:

```
for i in range(25):
```

Another common question is "Does the control variable have to be named `i`?" The answer is no. You should avoid using a name that you have already used earlier in your function—except perhaps another loop index—but otherwise any name is allowed, though the rules for good variable names still apply.

Remember that `range(25)` does indeed generate that 25-element list from 0 to 24.

5.2.3 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 actually *used* the variable in 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]:
```

This is perfectly fine for the cases where you want pure repetition, but sometimes the value of the control variable is important to the ongoing computation. 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):
    answer_so_far = 1
    for factor in range(1, n+1):
        answer_so_far = answer_so_far * factor
    return answer_so_far

```

When we call `factorial(4)`, the loop becomes:

```

for factor in range(1, 5):
    answer_so_far = answer_so_far * factor

```

What happens here?

- After the first time through the loop `answer_so_far` holds the result of its previous value (1) times the value of `factor` (1). When the loop completes its first iteration `answer_so_far` will be 1 (i.e. $1*1$).
- In the second iteration through the loop `answer_so_far` will again be assigned to hold the product of the previous value of `answer_so_far` times `factor`. Since `factor`'s value is now 2, `answer_so_far` will equal $1*2$, or 2, when this second loop iteration ends.
- After the third time through the loop, `answer_so_far` will be $2*3$ or 6, and the fourth time through `answer_so_far` will become $6*4$, or 24.

The loop repeats exactly 4 times because `range(1, 5)` has four elements: [1, 2, 3, 4].

Unrolled, this four-iteration loop created by `factorial(4)` becomes

```
# factorial function begins
answer_so_far = 1

# loop iteration 1
counter = 1
answer_so_far = answer_so_far * factor # answer_so_far will become 1
# iteration 2
factor = 2
answer_so_far = answer_so_far * factor # answer_so_far will become 2
# iteration 3
factor = 3
answer_so_far = answer_so_far * factor # answer_so_far will become 6
# iteration 4
factor = 4
answer_so_far = answer_so_far * factor # answer_so_far will become 24

# loop ends, 24 is returned
```

5.2.4 *Accumulating* answers

Consider what happened to `answer_so_far` throughout the iterations above. It started with a basic value and it changed each time through the loop, so that when the loop completed the returned `answer-so-far` was, in fact, the correct, final answer to the computation we had been seeking. This technique of *accumulation* is widespread, and the variable that approaches 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, `L`. Which variable is the accumulator here?

```
def listDoubler(L):
    dblList = []
```

```

for elem in L:
    dblList += [2*elem]
return dblList

```

```

>>> listDoubler([20, 21, 22])
[40, 42, 44]

```

In this case, the accumulator, `dblList`, is 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 Stroop Effect program, let's write a loop that calculates the average time difference between the control trials (displaying strings of X's) and the Stroop trials (displaying color words in different colors). Assume that the data for our trials, averaged over several users, is in two lists, named `controlData` and `stroopData`.

Each element in the list holds the mean time it took a user to respond to the control or Stroop strings, e.g., for three users the data might be

```

>>> controlData
[1.77, 2.42, 2.01]
>>> stroopData
[5.02, 4.92, 5.88]

```

There are many ways to approach this problem of finding the average differences between corresponding elements in these two lists. Here, to motivate another nuance of loops, we will

- Repeat the following for each user:
 - initialize a variable `totalDiff` to 0
 - find the difference between the control time and the Stroop time
 - add this difference to the value of the `totalDiff` variable
 - after the above loop ends, divide `totalDiff` by the number of trials.

How could we express this in Python? A first instinct might begin as follows:

```

totalDiff = 0
for time in controlData:
    # Get the corresponding time out of stroopData...

```

but we reach a dead-end at this point. We have the element from the `controlData` list, but we have no way of getting the corresponding element from the `stroopData` list. 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:

```

totalDiff = 0
for index in range(len(controlData)):
    totalDiff += controlData[index] - stroopData[index]
aveDiff = totalDiff / float(len(controlData))

```

We use the range command to force the loop control variable to iterate over the *index* of each term the list! With this `index` we can access the same position in both lists each time through the loop. Again, the accumulator idiom plays a central role: `totalDiff` starts at an appropriate basic value, 0, and the new differences are added in with each pass through the loop. In this case, additional work (the division) was needed to obtain the desired average. Note the `float` cast: this ensures that the division is floating-point, not integer.

Finally, this code makes a big assumption: it assumes that the lengths of the two lists are the same. What would happen if `controlData` were longer than `stroopData`? Or vice versa? How might we fix these problems? We leave these questions as food for thought. . . .

or *coffee* for thought!

5.2.5 Indefinite Iteration: while loops

In all of the examples above, we knew exactly how many times we wanted the loop to run. However, in many cases, we *can't* know how many iterations we need—how long we run the loop depends on some external factor out of our control.

Let's continue with our Stroop Effect analyzer. Above we assumed that the experiment would run for a specific number of trials. But what if the user gets the answer wrong? Should we count that trial, or ask the user to try again?

It seems clear that we shouldn't simply count that trial as correct, since if we did the user could "fool" the system by simply pressing the same button for every trial. We could simply ignore wrong answers, but then we would almost certainly end up with a *different* number of trials for each user of the program.

A reasonable approach is to throw out trials in which the user answers incorrectly. That way, we end up with a predetermined number of *correct* trials. Yet the user will need to respond to an *indefinite* number of color-string prompts, since neither we nor the user knows in advance how many incorrect answers will occur.

For-loops always run a definite number of times, so this situation calls for a different kind of loop: the *while-loop*. A while loop runs as long as its boolean condition is true.

Let's take a look at the structure of a while loop that implements the new desired behavior for our Stroop Effect program.

```
numCorrectDesired = 25
numCorrectSoFar = 0
data = []

while numCorrectSoFar < numCorrectDesired:
    startTime = getTime()
    correctColor = showNextWord()
    answer = raw_input()
    endTime = getTime()
    if isCorrect(answer, correctColor):
        data += [endTime-startTime]
    numCorrectSoFar = len(data)

print "Congratulations. You're done."
```

The while loop is similar to the for-loop in that it consists of a loop header (line 3) and a loop body (lines 4–9). The loop header consists of the following three elements, in order:

1. The keyword `while`
2. A boolean expression. In our example this expression is `numCorrectSoFar < numCorrectDesired`
3. A colon

As with for-loops, the loop body must be indented within the loop header. Thus, the congratulatory last line in the above example is *not* inside the loop body.

A while loop will execute as long as the boolean expression in the header evaluates to `True`. In this case, the loop will keep repeating as long as the length of the data array is less than 25. Notice that we've introduced a little more functionality to handle checking the user's answer. The `isCorrect` function takes the name of the displayed color (in `correctColor`) and the user's `answer` to see if user was correct.

We only add elements to the `data` list if the user's answer is correct, so the number of correct elements corresponds to the length of `data`. This is the line `numCorrectSoFar = len(data)`, which runs each loop iteration. As long as we don't have 25 correct trials, the loop will repeat. After the 25th correct response, the loop condition evaluates to `False`, and the loop halts. After the loop halts, any code after the loop begins to execute—the congratulatory message, in this example.

5.2.6 for loops vs. while loops

Often it's clear whether a computational problem calls for definite (`for`) or indefinite (`while`) iteration. In other cases you may feel less, well, *definite*.

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 `answer_so_far` 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 for a situation like this 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!

and we never get `n!` 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!` 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.

See if you can spot the error in the following version:

```
def factorial(n):
    answer = 1
    while n > 0:
        answer = answer * n
        n = n-1
    return answer
```

This code will also run forever. But why, when 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.

The Apple Corporation's
address is *One Infinite
Loop, Cupertino, CA*

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!

5.2.7 Creating infinite loops on purpose

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.

We're
pro-procrastination!

For example, consider a slightly modified version of the Stroop program. This one counts how many examples the user describes before making a mistake:

```
numCorrect = 0

while True:      # run forever -- or at least as long as needed...
    colorChosen = showNextWord()
    answer = raw_input()
    if isCorrect(answer, colorChosen):
        numCorrect += 1
    else:
        break

print "You got", numCorrect, "correct before you made a mistake."
```

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`?” After all, the loop above can be written with a more informative condition: Yes!

```
numCorrect = 0
mistake = False

while not mistake:
    colorChosen = showNextWord()
    answer = getUserAnswer()
    if isCorrect(answer, colorChosen):
        numCorrect += 1
    else:
        mistake = True

print "You got", numCorrect, "correct before you made a mistake."
```

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.

5.2.8 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 iMac, 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. Our recursive version runs out of memory before it’s able to complete 20,000 nested calls.

5.2.9 Assignment by *reference*

Should you choose to accept it

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 L holds the value of range(1000042) right now
M = L
```

This assignment makes L2 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 L aren’t lists themselves. . . .

How does Python make *both* integer and list assignments efficient? It does so through a simple, single rule for all of its datatypes:

Assignments copy a single reference.

Reference? Recall from Chapter 2 that all Python variables consist of both a reference, their memory address, and a value, the data that is referred to by that reference. We typically think only of the value of a variable, but you can also view the reference 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 (its memory location)
16793500     # this will be different on your machine!
```

Make a figure with box and arrows

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, 16793500 and then it gives x a copy of that
              # memory reference
>>> y = x     # this copies x's memory reference into y's

>>> id(x)    # asks for x's reference (its memory location)
16793500
>>> id(y)    # asks for y's reference (its memory location)
16793500
```

Make a figure with box and arrows

As you would expect, changes to `x` do not affect `y`:

```
>>> x = 43      # this puts 43 in the next memory slot, 16793488
>>> id(x)
16793488      # x's reference has changed
>>> id(y)
16793500      # but y's has not
```

Make a figure with box and arrows

Assignment happens the same way, regardless of the datatypes involved:

```
>>> L = [42,43] # this will create the list [42,43] and give its
                # location to L
>>> M = L      # give M that reference, as well

>>> L          # the values of each are as expected
[42,43]
>>> M
[42,43]

>>> id(L)      # asks for L's reference (its memory location)
538664
>>> id(M)      # asks for M's reference (its memory location)
538664
```

It is possible to reprogram how assignment works for user-defined datatypes, but this is the default.

Make a figure with box and arrows

As with integers, if an assignment is made, *one reference is copied*:

```
>>> L = [44]   # will create the list [44] and make L refer to it
>>> id(L)      # L's reference has changed
541600
>>> id(M)      # but M's has not
538664
```

Make a figure with box and arrows

For lists, however, our diagram is drawn to highlight an important difference between lists and integers: 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! We use a cloud to emphasize that the whole collection of references is our single list.

Because assignment copies only one memory reference, `L` and `M` contain *exactly the same set of references* as their elements!

This one-reference rule can have surprising repercussions. Consider this example, in which `x`, `L[0]`, and `M[0]` start out by holding the same reference to the value 42:

```
>>> x = 42     # to get started
>>> L = [x]    # similar to before
>>> M = L      # give M that reference, as well
```

```
>>> id(x)          # all refer to the same data
16793500
>>> id(L[0])
16793500
>>> id(M[0])
16793500
```

What happens when we change L[0]?

```
>>> L[0] = 43      # this will change the reference held by the
                  # "zeroth" element of L

>>> L[0]
43                # not surprising at all
>>> M[0]
43                # aha! L and M contain the SAME REFERENCES
>>> x
42                # x is a distinct reference

# Let's see

>>> id(L[0])      # indeed, the reference of that zeroth element
                  # has changed
16793488
>>> id(M[0])      # but the elements of M are
16793488
>>> id(x)         # x is still, happily, the same as before
16793500
```

5.2.10 Mutable datatypes can be changed using other names!

Note that M[0] has been changed in the above example, even though no assignment statements involving M[0] were run! Datatypes, such as lists, that allow assignment via shared references are called *mutable*. They are mutable, because their **components** can mutate “out from under them,” through another reference that points to the same component parts.

Datatypes that do not allow changes to a variable’s value—except by direct re-assignment via the variable’s name—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.

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:

```
>>> s = 'immutable'
>>> s[0] = ' '
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

*Two images
here—mutation can be
good... or bad.*

Python lets you compute what bits make up these datatypes, but you can’t change—or even read—the individual bits themselves.

Keep in mind that, mutable or immutable, you can always change the data to which a variable refers:

```
>>> s = 'immutable'
>>> s
'immutable'
>>> s = 'mutable'
>>> s
'mutable'
```

Take-home message

There are a few key ideas to keep in mind:

- All Python variables have both a reference and a value.
- Python assignment copies *one* reference.
- Mutable types allow assignment to component parts—this enables the value of a variable to change through other references that share its component parts.
- Immutable types do not allow assignment to component parts—but reassignment of the whole variable and its reference remains possible.

5.3 Taking advantage of mutable data: Sorting for Stroop

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. We have seen small examples of how mutable datatypes differ from immutable ones. In this section we return to our Stroop Effect example in order to motivate and illustrate a real example of data mutation and iteration together.

Imagine that we have collected data for our trials, and they reside in the lists `controlData` and `stroopData`. Now, we would like to sort these lists from smallest to largest. This will make the lists easier to analyze, for example, to find the median result among each dataset.

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.

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.

... and Don Knuth has said a lot of it!

- 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 *MinSort* because it proceeds by repeatedly selecting the minimum remaining element and moving it to the next position in the list. It is one variety of *Selection Sort*. *MaxSort* is another variety; it works by swapping the largest element remaining in each iteration.

What functions will we need to write *MinSort*? It seems we need only two:

- `index_of_min(L, starting_index)`, which will return the index of the smallest element in `L`, starting from index `starting_index`.
- `swap(L, i, j)`, which will swap the values of `L[i]` and `L[j]`

Here is the Python code for this algorithm:

```
def selectionSort(L):
    '''sorts L iteratively and in-place'''
    for starting_index in range(len(L)):
        min_elem_index = index_of_min(L, starting_index)
        swap(L, starting_index, min_elem_index)
```

Here is `index_of_min`:

```
def index_of_min(L, start_index):
    '''returns the index of the min element at or after start_index'''
    min_elem_index = start_index
    for i in range(start_index, len(L)):
        if L[i] < L[min_elem_index]:
            min_elem_index = i
    return min_elem_index
```

And `swap`:

```
def swap(L, i, j):
    '''swaps the values of L[i] and L[j]'''
    tmp = L[i]    # store the value of L[i] for a moment
    L[i] = L[j]   # make L[i] refer to the value of L[j]
    L[j] = tmp    # make L[j] refer to that stored value
```

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!

```
>>> stroopData    # Assume this list is already populated with data
[1.54, 0.43, 2.22, 2.12, 1.09]
>>> selectionSort(stroopData)
>>> stroopData
[0.43, 1.09, 1.54, 2.12, 2.22]
```

5.3.1 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 changing list elements through one variable changes the value of all of the variables. But where are the two variables referring to the same list? It seems that `stroopData` is the only variable referring to the original five-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.,

```
L = stroopData
```

occurs at the beginning of the call to `selectionSort(stroopData)`.

As a result, as long as `L` and `stroopData` refer to the same list, any changes that are made to `L`'s elements will affect `stroopData`'s elements—because `L`'s elements and `stroopData`'s elements are one and the same thing.

The take-home message here is that *passing inputs into a function is equivalent to assigning those inputs to the parameters of that function*. Thus, the subtleties of mutable and immutable datatypes apply, just as they do in ordinary assignment.

The following figure illustrates what is happening before and after the first call to `swap` in the above example.

FIGURE

When `swap` finishes, its variables disappear. But notice that even though `swap`'s `L`, `i`, `j`, and `tmp` are gone, the value of the list has been changed in memory. Because `swap`'s `L` and `selectionSort`'s `L` and the original `stroopData` 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!

5.3.2 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(L):
    '''sorts L iteratively and in-place'''
    for starting_index in range(len(L)):
        min_elem_index = index_of_min(L, starting_index)
        swap(L[starting_index], L[min_elem_index])    # now elements!

def swap(a, b):
    '''swaps the values of a and b'''
    tmp = a
    a = b
    b = tmp
```

The code looks almost the same, but now it does not work!

```
>>> stroopData      # Assume this list is already populated with data
[1.54, 0.43, 2.22, 2.12, 1.09]
>>> selectionSort(stroopData)
>>> stroopData
[1.54, 0.43, 2.22, 2.12, 1.09]
```

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 `L` or, to say the same thing, nothing happens to the elements of `stroopData`.

Figure 42? illustrates what is happening before and after the call to `swap`
Figure 42?

What happened? This time, `swap` has only two parameters, `a` and `b`, whose references are copies of the references of `L[start_index]` and `L[min_elem_index]`, respectively. Keep in mind Python's mechanism for assignment:

Assignments make a copy of a *single* 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 `L` have not changed. Thus, the value of `L` does not change. Since the value of `stroopData` *is* the value of `L`, nothing happens.

5.3.3 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 3 where lists were used to store strings, numbers, and combinations of those datatypes. 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

```
>>> A = [[5,6],[7,8]]
```

creates a list named `A` where each of `A`'s elements is itself a list. Figure 5.9 illustrates this 2D array graphically.

You can access the individual elements in the 2D array by using nested indices. For example:

```
>>> A[0]          # Get the first element of A
[5,6]
>>> A[0][0]      # Get the first element of A[0]
5
>>> A[1][1]      # Get the second element of A[1]
8
```

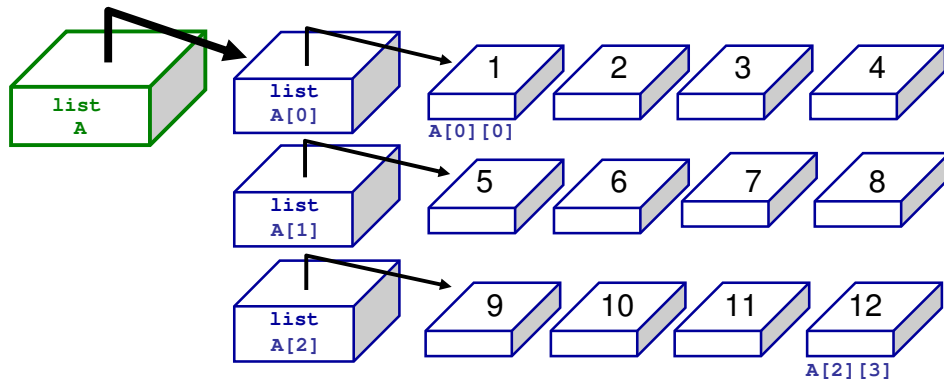



Figure 5.9: A graphical depiction of a 2D array. Following our model from above, the green box is stored in the CPU, and the rest of the data (including the references to other lists) is stored in memory.

We can also ask questions about A’s height (the number of rows) and its width (the number of columns):

```
>>> len(A)      # The number of elements (rows) in A
3
>>> len(A[0])  # The number of elements in a list in A (i.e.,
                # the number of columns)
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. Returning to our Stroop example, we can immediately see an application: a 2d array could hold all of the raw timing data from the many trials run by many users. Each row could represent a single user’s data; each column would represent an individual trial.

You could then analyze that data, e.g., to find the mean response time for each user individually:

```
def findMeans(data):
    '''findMeans(data): returns a list where each element
       in the list is the mean for a particular user’s response
       time.
       input data: a 2D array where each row is data from a
                   single user
    '''
    means = []
    for dataset in data:
        total = 0.0
```

```
    for elem in dataset:
        total += elem
    means += [total/len(dataset)]
return means
```

In the above code, the outer loop controls the iteration over the individual users. That is, the value of `datasets` is an entire list each iteration of the for loop. The inner loop then iterates over the elements in an individual user's data. In other words, `elem` is simply a number.

We can also write the above code as follows:

```
def findMeans(data):

    means = []
    for user in range(len(data)):
        total = 0.0
        for datapoint in range(len(data[user])):
            total += data[user][datapoint]
        means += [total/len(data[user])]
    return means
```

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.

If we have the user's data stored as a 2D array, it's also easy to analyze the data in the other dimension. For example, we can calculate the average time that all users took on each individual trial (i.e., the average across users on the first trial, the second trial, etc):

```
def meansOfTrials(data):
    '''meansOfTrials(data): returns a list where each element
        in the list is the mean for a particular trial across
        users.
        input data: a 2D array where each row is data from a
                    single user
    '''
    means = []
    if len(data) == 0:
        return means
    for datapoint in range(len(data[0])):
        total = 0.0
        for user in range(data):
            total += data[user][datapoint]
        means += [total/len(data)]
    return means
```

Notice that this code is extremely similar to the code above, but we've reversed which loop is on the outside. Now the outer loop iterates over the data points (columns)

and the inner loop iterates over the user. Because the inner loop runs completely through from beginning to end every time the outer loop runs, the inner loop will sum over all users for a single data point.

5.3.4 Dictionaries

So far we've looked at one mutable datatype: 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.

To motivate the need for a dictionary, let's once again return to the Stroop Effect program. During the trials, the user is shown a colored string (either X's or color names) and asked to respond with the color of the string. Above we also included a function that would determine if the user had entered the correct key for the color that was displayed. We did not write that function, but let's go ahead and write it now:

```
def isCorrect(answer, colorChosen):
    '''Given the user's answer and the color displayed, return True
       if the user entered the correct keystroke for the color.
       input: answer: a string representing the key the user pressed
              colorChosen: a string (all lower case) representing
                           the color displayed.
    '''
    if colorChosen == 'green' and answer == 'd':
        return True
    if colorChosen == 'red' and answer == 'f':
        return True
    if colorChosen == 'black' and answer == 'j':
        return True
    if colorChosen == 'yellow' and answer == 'k':
        return True
    return False
```

This function will work, but you might already but uncomfortable with it (and if you're not yet, you will be!). First, it seems tedious to have to list each color and key in a separate if statement (yes, we could combine them all into 1 if statement, but it would be a LONG if statement). However, the more pressing concern is that the mapping between colors and keys is hard-coded in this function. In our example, we arbitrarily chose a mapping between keys and colors to try to make it easy for users to reach all of the color keys. However, this mapping might not be one we permanently want to use. What if we change the colors we're using? Or what if we decide it would be a better idea to simply map each color to the first letter of its name? We would have to go into this function and change the mapping by changing the code.

This might not seem like a big deal, but what if we rely on these mappings in other places too? (For example, the code that displays the instructions to the user certainly needs to know the mapping from colors to keys to communicate them to the user). If

one mapping or color changed, we would have to go through and find all of the places in the code that needed to be changed. This process would be tedious and it's likely that we would miss at least one place, leading to hand-to-find errors.

In short, succinct design is important!

What we need is a single place to store this mapping 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:

```
>>> d = {}          # creates a new empty dictionary
>>> d['green'] = 'd'      # create an association between
                        # 'green' and 'd'
                        # 'green' is the key and 'd' is the value
>>> d['red'] = 'f'       # ditto for 'red' and 'f'
>>> d                  # display the mappings currently in
                        # the dictionary
{'green': 'd', 'red': 'f'}
>>> d['red']           # get the item mapped to 'red'
'f'
>>> d['f']             # get the item mapped to 'f'. Will
                        # cause an error because mappings are
                        # one-way--'f' is not a valid key.
Traceback (most recent call last):
  File "<pyshell#14>", line 1, in <module>
    d['f']
KeyError: 'f'
>>> d.has_key('f')    # Check whether a key is in the
                        # dictionary
False
>>> d.keys()          # Get the keys in the dictionary
['green', 'red']
>>> d[1] = 'one'       # keys can be any immutable type
>>> d[1.5] = [3, 5, 7] # values can also be mutable, and
                        # we can mix types in the same
                        # dictionary
>>> d
{1.5: [3, 5, 7], 1: 'one', 'green': 'd', 'red': 'f'}
>>> d[[1, 2, 3]] = 'one' # Keys cannot be mutable
Traceback (most recent call last):
  File "<pyshell#36>", line 1, in <module>
    d[[1, 2, 3]] = 'one'
TypeError: list objects are unhashable
>>> keyMap = {'green': 'd', 'red': 'f', 'black': 'j', 'yellow': 'k'}
>>> keyMap            # a shorthand way to create a dictionary
{'black': 'j', 'green': 'd', 'yellow': 'k', 'red': 'f'}
```

The order in which the key-value pairs appear in Python’s output of `keyMap` represents its internal representation of the data.

Now let’s look at how adding a dictionary simplifies our Stroop Effect code:

```
def isCorrect(answer, colorChosen, keyMap):
    '''Given the user's answer and the color displayed, return True
       if the user entered the correct keystroke for the color.
    input: answer: a string representing the key the user pressed
           colorChosen: a string (all lower case) representing
                       the color displayed.
           keyMap: A dictionary mapping colors to keystrokes.
    '''
    return keyMap[colorChosen] == answer
```

Pretty simple... and flexible too!

5.4 Displaying Colored Strings (Randomly)

We’ve almost got all the pieces we need to build our Stroop Effect program. The last major piece we are missing is the ability to display colored strings to the user.

There are many ways to accomplish this task. In this section we will look at only one: a graphical package for Python called `vPython`. `vPython` is introduced more thoroughly in the next section. Here we will use it only for its display abilities.

First, to use `vPython`, we have to include it in our program by placing the following line at the top of our file:

```
from visual import *
```

Now, creating colored strings in `vPython` is simple. Strings in `vPython` are called labels, and to create and display a new label all you have to do is type:

```
>>> displayLabel = label(text='Test', height=24, box=0,
                        opacity=0, color=color.black)
```

You will see a new window pop up with a text string in the window that says “Test”. The `label` function is similar to, but not the same as functions we’ve seen before. First, it’s not a “normal” function, but rather a special function called a *constructor* that creates a new *object*. Briefly, objects are simply custom data structures, like lists and dictionaries, but defined by the user. We’ll talk much more about objects in the next chapter, so don’t worry too much about their details right now. Second, the arguments to the label constructor look a bit strange: Why do they involve equals signs? This is because the label constructor takes a number of *optional arguments*. That is, we can choose to pass in these arguments or not. If we don’t pass them in, Python will simply use a default value for each argument. Because we are not required to include all the arguments to this function, we have to tell Python which value goes with which argument, hence the equals signs.

Unfortunately, the background for the window that pops up is also black by default, making the text hard to read. To change the background color, we need to use the line:

In fact, the label constructor takes many more arguments than we are passing here, all of which are set to their default values.

How Color Is Represented on the Computer

A description of how color is represented on the computer.

```
>>> scene.background = color.white
```

This syntax probably looks especially funky, but it won't after you finish the next chapter. For now all you need to know is that `scene` is a variable that holds the window that is created when we first make a new label. It is a *global variable*, which means that we can access it from any function. It is automatically created by vPython when the first object is created (notice that we did not have to set its value). To change the background color, we set its `background` property. `color.white` is a representation of the color white that is built-in to vPython. vPython also has built-in representations for other common colors, including black, red, blue, yellow, green, orange, etc.

After the initial setup of the program, we would like to clear the text label from the screen. In vPython we do this by simply making the label invisible as follows:

```
>>> displayLabel.visible = False
```

To display new text strings in different colors, we just use the line above, substituting the word "Text" for the text we want to display and the color for the color we would like to use for the text.

The final piece in our puzzle is to somehow randomize the order in which we display the colored strings. Let's assume we have a list of colors, e.g.:

```
>>> listOfColors = [color.red, color.blue, color.black, color.yellow]
```

and we would like to choose one color randomly from the list. To do this we can use the `random` package which is built in to Python. As usual when using a package, we need to include a line at the top of our file:

```
from random import *
```

Then we can use any function that is available in that package. In this case we want the `choice(L)` function. This function will choose a member of this list `L` at random and return it. To display a string of `X`'s in a randomly chosen color we could write:

```
>>> chosenColor = choice(listOfColors)
>>> displayLabel = label(text='XXXXX', height=24, box=0, opacity=0,
                        color=chosenColor)
```

A side note: it is not ideal to force the user to press **Enter** after each response. However, because this requirement is constant in all of our trials, it should not affect the overall experiment results. There are other ways of collecting user input, which we will learn about later, that solve this problem. —*perhaps vPython's event-handling?*

This figure will list the data and computation that the program will perform.

Figure 5.10: Data and computation in the Stroop Effect program

This figure will give variable names and data types for the data listed in the above figure.

Figure 5.11: Variables in the Stroop Effect program

5.5 Putting It All Together: Program Design

Now that we've got all the tools, it's time to construct the whole Stroop Effect program. However, as a disclaimer, large scale program design can at times feel more like an art than a science. 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.

Maybe more science than art!

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. Figure 5.10 shows one possible list of the data our program must keep track of and the computation it must perform.

CS is at least as creative and precise as careful writing; don't be afraid of creating—and redoing—several drafts!

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 pieces of data listed in Figure 5.10. Our answers are shown in Figure 5.11.

Finally, now that we have identified our data and the computation that our program needs to perform, we 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.

5.5.1 A program to quantify the Stroop Effect

Listing 5.1, on the following page, gives a complete solution.

Listing 5.1: A Stroop-Effect Program

```
from visual import *
from random import *
from time import *

def runStroopTest():
    '''runStroopTest(): controls the overall execution
       of the Stroop Test program.'''
    NUM_TRIALS = 25

    nonTextData = []
    textData = []

    nonTextStrings = ['xxxxx']
    keyMap = {'blue': 'd', 'red': 'f', 'black': 'j',
              'yellow': 'k'}
    colorMap = {'blue': color.blue, 'red': color.red,
                'black': color.black, 'yellow': color.yellow}

    displayLabel = label(text='Test', height=24, box=0,
                          opacity=0, color=color.black)
    scene.background = color.white

    printInstructions(keyMap)

    displayLabel.visible = False

    textSecond = choice([True, False])
    if textSecond:
        nonTextData = runTrials(NUM_TRIALS, nonTextStrings,
                                colorMap, keyMap)
        textData = runTrials(NUM_TRIALS, keyMap.keys(),
                              colorMap, keyMap)
    else:
        # switch the order to get balance
        textData = runTrials(NUM_TRIALS, keyMap.keys(),
                              colorMap, keyMap)
        nonTextData = runTrials(NUM_TRIALS, nonTextStrings,
                                colorMap, keyMap)

    analyzeAndDisplay({'nonTextData': nonTextData,
                      'textData': textData})
```



```
def analyzeAndDisplay(dataSets):
    '''Analyze sets of data points
       input:
           datasets: a dictionary mapping from dataset name
                     (string) to dataset (list).'''
    # Right now this function simply prints the mean and
    # standard deviation of each data set. However, in the
    # future this could do many more exciting things like fit
    # functions to the data and plot them or aggregate data
    # over many users (that might be better in a separate
    # function).
    for dsname in dataSets.keys():
        print dsname + ":"
        print "\taverage time: ", ave(dataSets[dsname]), \
            "seconds; st dev:", stdev(dataSets[dsname]), \
            "seconds"

def ave(L):
    '''ave(L): returns the average of a list of numbers.
       input L: a list of numbers'''
    return sum(L) / float(len(L))

def stdev(L):
    '''stdev(L): returns the standard deviation of a list
       of numbers.
       input L: a list of numbers'''
    m = ave(L)
    total = 0
    for num in L:
        total += (num-m)**2
    total = total / float(len(L)-1)
    return math.sqrt(total)
```

```
def remove(L, item):
    '''remove(L, item): return a list containing all the
        elements of L except for the first occurrence of item.
        Does not change L.
        input L: a list
        input item: an element that may or may not be in
            the list'''
    retL = []
    found = False
    for elem in L:
        if elem == item and not found:
            found = True
        else:
            retL += [elem]
    return retL

def printInstructions(colors):
    '''printInstructions(colors): prints the instructions for
        the Stroop test trials.
        input colors: a dictionary mapping color strings
            to key strokes'''

    print "Please position the display window so that it is"
    print "next to this window, where you can see both"
    print "windows easily. The display window may be"
    print "'hiding' behind the other windows."
    raw_input("Press <Enter> when you are ready"
              " to continue...")

    print "\n\n"
    print "In this experiment you will be shown strings of"
    print "text in different colors You will be show the"
    print "strings in two rounds. In one round, the text"
    print "strings will spell words. In the other, it will"
    print "just be Xs. For each string you see, please input"
    print "the COLOR of the text as fast as you can, using"
    print "the following key mapping:"
    for ks in colors:
        print "\tPress", colors[ks], "then <Enter> for", ks

    print "\n\n"
    print "In each case, please work as quickly as possible,"
    print "trying not to make mistakes. If you make a"
    print "mistake, you will be shown another string."
```

```
def runTrials(numTrials, strings, colorMap, keyMap):
    '''runTrials(numTrials, strings, colorMap, keyMap): Run
       the trials for the stroop test and return the time data
       collected from the trials, as a list.
       input:
           int numTrials: the number of trials to run
           list strings: a list of strings to display for
                       the text
           dictionary colorMap: a dictionary that maps color
                               strings to colors
           dictionary keyMap: a dictionary that maps color
                               strings to keystroke strings'''

    i = 0
    raw_input("\n\npress <Enter> when ready"
              " to begin round...")
    data = []

    while i < numTrials:
        c = choice(strings)
        colorsRemove1 = remove(colorMap.keys(), c)
        cColor = choice(colorsRemove1)
        beforeTime = time()
        displayLabel = label(text=c, height=24, box=0,
                             opacity=0,
                             color=colorMap[cColor])

        answer = raw_input()
        afterTime = time()
        displayLabel.visible = 0
        if answer == keyMap[cColor]:
            data += [afterTime-beforeTime]
            i += 1
        else:
            print "INCORRECT. Try Again"
    print "Round over"
    return data
```


Chapter 6

OOPs! Object-Oriented Programs

I paint objects as I think them, not as I see them. —Pablo Picasso

Imagine that you're a rocket scientist working on a project to launch a rocket to the planet Spamiter. You have been assigned the task of testing some software that determines whether the rocket has sufficient fuel to perform a mission-critical maneuver. As you test the software, you notice that it often reports that there is not enough fuel to perform the maneuver when, in fact, you are certain there is 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 and 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 while `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 fail to perform its mission.

When you print the values of `fuelNeeded`, `tank1`, `tank2`, and `tank1 + tank2` you see the problem:

```
>>> fuelNeeded
0.042000000000000003
>>> tank1
```

This material, as cool as it is, is not rocket science.

Bummer!

```
0.035999999999999997
>>> tank 2
0.00600000000000000001
>>> tank1 + tank2
0.041999999999999996
```

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 computer can store, resulting in the kinds of unexpected behavior that we've seen here. A rational thing to have!

Or even a *fraction* of them all.

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 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 comparing the numerators and comparing the denominators, which involves comparing integers and is thus free from numerical imprecision.

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. By the end of this chapter, you will be a master of object-oriented programs (OOPs). You will be able to define your own types of data and then use them in programs that are substantially easier to read and write than without OOPs.

6.1 A Rational Solution

Let's get started by defining a rational-number type. To do this, we build a Python "workshop" for constructing rational numbers. This workshop 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 in a moment and 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” new rational numbers to our heart’s content. Here’s an example of calling this “workshop” to manufacture 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 manufactures 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 selfish 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* inputs (or “arguments”) whereas the instruction `r1 = Rational(36, 1000)` has only supplied *two* inputs. That’s weird, but—as you may have guessed—the first input argument is passed in automatically by Python and it is a reference to the new empty `self` object that we’ve just had manufactured 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 for all of these variables.

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
>> r1.denominator
1000
```

The “dot” in `r1.numerator` is saying, “go to the object called `r1` and look at the variable named `numerator`.” We used the “dot” in the function `__init__` as well:

`self.numerator = num`. The “dot” was doing the same thing there. It said, “go into the `self` object and look at the variable named `numerator`.” (It may seem strange to you that in the latter case, we were telling Python to “look” at a variable that didn’t exist yet. But this is just Python being Python. Remember, if you have a statement like `myNum = 42` then if `myNum` has been defined earlier, its value is just changed. However, if this is the first mention of `myNum` then Python gladly creates a new variable called `myNum`.) Figure 6.2 shows the situation now.

Figure 6.2: `r1` refers to the `Rational` object with its `numerator` and `denominator`.

In our earlier example, we “called” the `Rational` “workshop” twice to make two different rational numbers in the example below:

```
r1 = Rational(36, 1000)
r2 = Rational(6, 1000)
```

The first call, `Rational(36, 1000)` manufactured 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 variable 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 workshop—technically known as a *class*—called `Rational`. This `Rational` class describes a template for manufacturing a new type of data. That workshop can now be used to manufacture a multitude of items—technically known as *objects*—of that type. Each object will have its own variables, in this case `numerator` and `denominator`, each with their own values.

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.

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:

This would be a short chapter if that was the whole story!


```
>>> 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 $r1 + r2 = \frac{ad+bc}{bd}$. 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.

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 `Blah` followed by all of the `N` arguments that are passed in explicitly. Although we've been calling this first implicit argument "`self`", this is not necessary. It can be called anything at all. 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!

Now, for example, we could type:

"Snazzy" is a technical term.

```
>>> r3.numerator
>>> r3.denominator
```

What 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.

6.2 Overloading

“Spiffy” is yet another technical term.

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. 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 is “good” overloading. “Bad” overloading involves more than 18 units.

This feature is called *overloading*. 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 `Rationals` 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 us 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.

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 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://www.Python.org/doc/2.5.2/ref/customization.html>.)

For example, we can overload the `>=` symbol by defining a method called `__ge__` (which stands for **g**reater **t**han or **e**qual). Just like `__eq__`, this method takes two arguments: A reference to the calling object that is passed in automatically (e.g. `self`) and a reference to another object to which we are making the comparison. So, we could write our `__ge__` method as follows:

```
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:

```
>>> 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!

This example serves to doubly underscore the beauty of overloading!

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.

6.3 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(3.141)
'3.141'
```

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.

6.4 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 the previous chapter.) *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(input):
    input.numerator += 1
```

What happens when we invoke function `foo`? Notice that function `bar` is not returning anything. However, the variable `input` 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 `numerator` of 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 `input` is referring to that memory location. When Python sees `input.numerator += 1` it first goes to the memory address referred to by `input`, 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.

You might say that Python is “class”y! 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.

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.

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 `appends` 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.

6.5 Why are OOPs Important?

This is all *cool*, but why is object-oriented programming *important*? That's a fair question. As we've seen 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.

"Yucky" is *not* a technical term.

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.

Is object-oriented programming just used to represent new numerical types such as rational numbers? Not at all! Virtually any program that has different logical parts can benefit from the object-oriented design and programming. Indeed, we'll soon see an example in 3D graphics. In fact, although we've been talking about object-oriented *programming*, the truth is that "object-oriented" is really a design principle.

But the acronym "OODs" is nowhere near as good "OOPs"!

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.

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.

6.6 Getting Graphical with OOPs

Warning! This section contains graphical language!

We've seen that object-oriented programming is cool and, hopefully, you now believe that it's useful. If not, this section should convince you! Next, we're going to look at how object-oriented programming gets used in computer graphics. If you wish, you can even write your own very cool 3D graphics programs in Python as you read this section.

Aye, yay, yay! (Or Eye, eye, eye!)

Computer graphics offers a particularly compelling domain where object-oriented programming is truly the secret to all happiness. Before we get into the nuts-and-bolts, let's stop and do a quick thought experiment. Imagine that you're a computer graphics animator for the upcoming video game *Attack of the Three-Eyed Aliens*. Imagine that the characters in this animation each have a face with three eyes, a nose, two ears, and a mouth with upper and lower lips. Notice that the face has three eyes. If we've defined an `Eye` class, we can simply make three instances (objects) of this class rather than define each eye from scratch. Similarly, we might define an `Ear` class and a `Lip` class. Then, the `Face` class could be defined using these "lower level" classes and might look something like this:

```
Class Face:
    def __init__(self):
        self.leftEye = Eye()
        self.middleEye = Eye()
        self.rightEye = Eye()
        self.leftEar = Ear()
        self.rightEar = Ear()
        self.upperLip = Lip()
        self.lowerLip = Lip()
```

Let's Face it, that's slick!

Now, if you wanted to twitch just the left ear, you could make a change that one object in the face without having to think about the rest of the face. Moreover, as you defined your swarm of thousands of characters, each one of them would have a face that is simply an instance of this `Face` class.

Time to get real; let's do some 3D graphics! Python has many wonderful add-on "modules" that allow it to do all kinds of amazing things. One of these modules, called *VPython*, allows you to use Python to write interactive 3D graphics programs. Of course, video games leap to mind as an obvious application, but scientists use VPython to write scientific visualizations (for example to understand the dynamics of interacting atoms, molecules, or planets), and there are numerous other places where 3D graphics are fun and useful. If you search for "VPython" in your favorite search engine, you'll find the link. From there, you can download VPython to your own computer (it's free) and find great documentation.

In this section, we'll look at some of the highlights of VPython and, particularly, its connection to object-oriented programming. Using a computer that has VPython

installed, we can type the following two lines:

```
>>> from visual import *
>>> b = box()
```

The first line is simply importing everything from VPython’s `visual` package that support 3D graphics. The second line is the interesting one. VPython has a class called `box`, and we are asking for an object of type `box` and assigning `b` to refer to it.

If all goes well on your computer, a window will pop up that looks like a gray square. It’s actually a 3D box, it’s just that we’re looking at it face-on. You can rotate and zoom in and out as follows: In the display window, 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 zoom in or out of the scene (on a 2-button mouse, hold down the left and right buttons; on a 1-button mouse, hold down the Option key).

Just like our `Rational` class had *attributes* such as `numerator` and `denominator`, the `box` class has attributes as well. One of them is called `color`. Recall that you can get the value of an attribute by typing the name of the object followed by a dot followed by the name of the attribute:

```
>>> b.color
(1.0, 1.0, 1.0)
```

Huh?! 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. If you want to make the color bright red, you could type

```
>>> b.color = (1.0, 0.0, 0.0)
```

Play with this a bit and let us know when you’re ready to move on.

The `box` class has other attributes too. One of them is called `pos` and, you guessed it, it stores the position of the box in 3 dimensions. The coordinate system used by VPython is what’s called a “right-handed” coordinate system: If you stick out your thumb, index finger, and middle finger so that they are perpendicular to one another, 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 starting rotating around 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.

Take a look at the position of the box by typing `b.pos`. You’ll notice that you see the following:

```
>>> 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. See! We told you classes were useful. “OK,” we hear you concede grudgingly, “but what’s the point of a `vector` (no pun intended)?”

We’ll wait. Maybe.

If your roommate sees you staring at your fingers, just explain that you are doing something very technical.

Is vector class also called linear algebra?

Take a look at these 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! You can size up these attributes in each of the three directions on the VPython web site too.

The point is that the vector class has some methods defined in it for performing vector operations, but let's not worry about that now. However, we can now move our box like this:

```
>>> b.pos = vector(0, 1, 2)
```

Boxes have other attributes too. For example the `size` attribute is a vector that indicates the size of the box and the `up` attribute is a vector that indicates which way is "up." By the way, VPython has lots of other shape classes including spheres, cones, cylinders, and many more. 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. You'll see on the VPython web site that we can use degrees instead and that rotation can be set to occur about any axis.) What is `rotate` actually doing? It is simply applying a transformation that changes the attributes of the box so that it is in the right position and orientation! In other words, when an object invokes its `rotate` method, that method recalculates the attributes of the object and then VPython simply renders the object on the screen. Try doing a rotation and looking at the object's `up` attribute afterwards.

"Render" is just a fancy way of saying "draw."

Finally, let's put this all together to write three short VPython programs. First, let's write a *very* short program that rotates a red box forever:

```
from visual import *

def spinbox():
    myBox = box()
    myBox.color = (1, 0, 0)
    while True:
        # Slow down the animation to 60 frames per second.
        # Try removing this line or changing 60 to something else.
        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(0, 10):
        x = random.randint(-5, 5) # random integer between -5 and 5
```

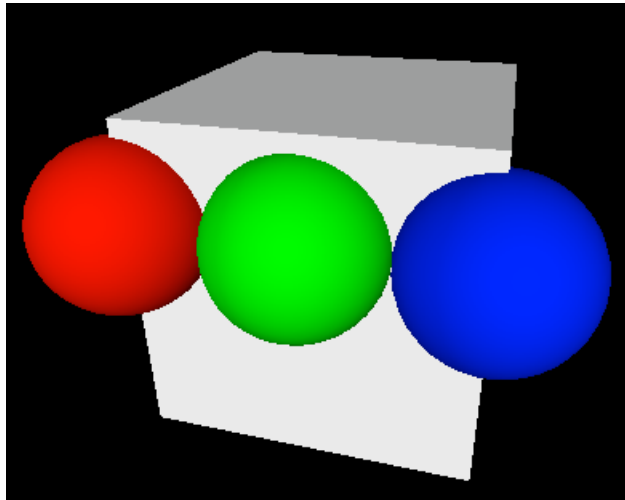


Figure 6.4: A weird “face” shape made of a box and three spheres.

```

y = random.randint(-5, 5)
z = random.randint(-5, 5)
red = random.random()      # random number between 0 and 1
green = random.random()
blue = random.random()
newBox = box()
newBox.pos = vector(x, y, z)
newBox.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.

Finally, what if we want to define a new kind of shape? We can define a class for this new shape and then make as many objects of this shape type as we like. For example, let’s write a class for a simplified version of the face class for the video game that we mentioned at the beginning of this section. For the sake of simplicity, we’ll just have a block head and three eyes, as shown in Figure 6.4. (You are, of course, welcome to add the lips and ears on your own!)

We’ll define the `Face` class using `box` and three `spheres`, and then we’ll put these body parts into a list called `parts` just as we put cubes into a list in the previous example. Putting the parts into a list allows us to manipulate them more easily. For example notice that we have also defined a `move` method that takes `x`, `y`, `z` as inputs and shifts (or “translates”) the face from its current position by the vector `(x, y, z)`.

Here we only pay lip service to additional features.

By placing all of the body parts into a list, the `move` method simply uses a `for` loop to change the `pos` of each body part.

```
from visual import *

class Face:
    def __init__(self):
        self.head = box()          # box head
        self.leftEye = sphere()    # left eye
        self.middleEye = sphere() # middle eye
        self.rightEye = sphere()   # right eye

        self.leftEye.color = (1, 0, 0) # the left eye is red...
        self.leftEye.radius = 0.25     #... and has a small radius...
        self.leftEye.pos = vector(-0.5, 0.2, 0.5) #... move it here

        self.middleEye.color = (0, 1, 0) # the middle eye is green...
        self.middleEye.radius = 0.25
        self.middleEye.pos = vector(0, 0.2, 0.5)

        self.rightEye.color = (0, 0, 1) # The right eye is blue...
        self.rightEye.radius = 0.25
        self.rightEye.pos = vector(0.5, 0.2, 0.5)

        # Finally, all of the body parts go into a list
        self.parts = [self.head, self.leftEye, self.middleEye,
                      self.rightEye]

    def move(self, x, y, z):
        moveVector = vector(x, y, z)
        for part in self.parts:
            part.pos = part.pos + moveVector
```

Before we let you go hang out with your friends, or eat a pizza, or write the complete implementation of *Attack of the Three-Eyed Aliens*, let's just point out the loveliness of this object-oriented idea. By defining classes, we can provide the user of the class with an abstraction that allows her or him (and, in fact, it may be *you*) to just think about the important things (like being able to add rational numbers or move a funny-looking face). Indeed, we can build a hierarchy of abstraction: a character in our video game could be a class that is now defined in terms of a face and other parts. A parade could be defined as a list of these characters along with other attributes, and so on and so forth.

All right, enough, go have your pizza!

Index

- `__eq__`, 137
- `__init__`, 133
- `__str__`, 138
- 2D arrays, **118**

- accumulation, **106**
- accumulator, **106**
- adder
 - full, **74**
- Algorithms, **2**
- analog computers, 63
- AND, **69**
- AND gate, **72**
- arguments, **21**
- arrays, **118**
- ASCII, **67**
- assembly language, **81**
- assignment statement, **19**
- Attack of the Three-Eyed Aliens, **142**
- attributes, **143**

- base 10, **65**
- base 2, **65**
- base case, **34**
- Binary, **65**
- bits, **65**
- body, **104**
- Boole, George, 69
- Boolean algebra, **69**
- Boolean expression, **26**
- byte, **68**

- central processing unit (CPU), **77**
- class, **134**
- comment, **14**
- comparison operators, **27**
- compiler, **92**
- compression, 68

- conditional jump, **85**
- conditional statement, **26**
- constructor, **123**

- docstring, 14

- four-color problem, **30**
- functions, **14**

- global variable, **124**

- header, **103**

- immutable, **114**
- infinite loop, **110**
- instruction register, **78**
- interpreter, **92**
- iteration, **100**
- iterative, **102**

- jump
 - conditional, 85
 - unconditional, 84

- keys, **122**
- keyword, **21**
- Knuth, Donald, 5

- layer of abstraction, **141**
- logic symbols, 73
- loops, **100**

- MaxSort, **116**
- method, **134**
- MinSort, **116**
- minterm expansion principle, **71**
- modular design, **141**
- modularization, **16**
- mutable, **114**

- no-operation, **88**
- NOT, **70**
- number representations
 - sign-magnitude, **66**
- numerical imprecision, **132**
- nybble, **68**

- object, **123**
- object-oriented programming, **132**
- objects, **134**
- opcode, **78**
- operation code, **78**
- optional arguments, **123**
- OR, **70**
- OR gate, **72**
- overloading, **136**

- parameter passing, **87**
- PC, **79**
- Picobot, **5**, 5–12
- pixel, **68**
- program counter, **78**

- Random Access Memory (RAM), **76**
- recursive definition, **33**
- registers, **77**
- ripple-carry adder, **74**

- S-R latch, **76**
- scope, **33**
- Selection Sort, **116**
- Shannon, Claude, **69**, **72**
- sign-magnitude, **66**
- signature, **21**
- stack, **32**, **88**, 88–90
 - pop, **89**
 - push, **89**
- stack discipline, **90**
- stack pointer, **89**
- state, **7**
- statements, **18**
- strings, **68**
- Stroop Effect, **98**
- Stuff, **1**, **4**

- truth table, **69**
- two's complement, **66**

- unconditional jump, **84**
- UTF, **68**
- UTF-8, **68**

- values, **122**
- Variables, **18**
- von Neumann architecture, **78**
- von Neumann, John, **78**

- while-loop, **108**

- zoogenesis, **2**