CHAPTER 1

Introduction

1.1 Objectives

- To provide examples of computer science in the real world
- To provide an overview of common problem-solving strategies
- To introduce Python's numeric data types
- To show examples of simple programs
- To introduce loops and simple functions
- To introduce turtle graphics

1.2 What Is Computer Science?

Computers are a vital part of our lives. They help to control the planes we fly in and the cars we drive in. They keep track of the buying and selling of stocks in financial markets around the world. They control complex surgical machinery and the pacemakers that are embedded in our bodies. Computers help us to communicate quickly and efficiently with others throughout the world. If you are reading this book, it is likely that you have used a personal computer for writing a paper, surfing the Web, or sending an email. You may have more experience. Perhaps you have hooked up a home network, or even built your own web page. However, even though computers have become appliances that are part of the backdrop of our lives, the chances are that you have not explored the world of programming.

The important question that we must answer in this chapter is "what is computer science?" The problem with answering this question is that the term is misleading right from the start. A look in a dictionary will tell you that science is the theoretical and experimental

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION

study of the natural world. It seeks to understand how things work by forming hypotheses, conducting experiments, and analyzing results. Given that definition, you might think that computer science is the exploration and discovery of how a computer works. As a novice, that might be interesting to you, but as a discipline it makes little sense given that computers are manufactured by humans. Computer science is not like biology or physics, disciplines where we are trying to understand the working of the human body, or how the universe works. In the words of Edsgar Dijkstra: "Computer science is no more about computers than astronomy is about telescopes."

So, what is computer science? **Computer science** is the study of **algorithms**.

To put it another way, computer science is primarily about problem solving and computational process. For beginning computer scientists, finding the solution to a problem is often the easiest part. Turning the solution into a set of step-by-step instructions that can be performed by a computer (creating a computational process) is often difficult. Computer scientists often call this set of instructions a **program**. You may think of it as something like a recipe for a beginning cook. First, bring the water to a boil, then add the macaroni, and so on.

Since you are familiar with using advanced programs that are designed to make the computer look intelligent, it is important to dispel that idea right away. Here are six important things to remember about computers as you are learning to program:

1. Computers are dumb.
2. Computers only do what you tell them to do.
3. Computers do what you tell them to do really fast, so they appear smart (but they are not).
4. Computers don't remember anything unless you tell them how to remember.
5. Computers take your instructions literally. If you tell them to do something dumb, they do it.
6. A computer only does what it is told and in exactly the order you tell it.

1.3 Why Study Computer Science?

Now that you have a better idea of what computer science is, you may be wondering why you should learn any more about it. Our belief is that computer science is for everyone. Some of the biggest computer success stories come from people who were not even computer

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION

scientists by training but came to computer science later in their careers because they had an interesting problem to solve. For example, a physicist by the name of Tim Berners-Lee at the European Laboratory for Particle Physics (CERN) needed a better way for physicists around the world to share information. The solution to this information-sharing problem became what is today called the World Wide Web. Berners-Lee wrote the computer programs for the first web server and browser. Today he is Director of the World Wide Web Consortium and a professor at the MIT Computer Science and Artificial Intelligence Laboratory.

1.3.1 Everyday Applications of Computer Science

Even if you know you want to be a computer scientist, the fact is that few computer scientists work only on problems limited to computers such as building a better operating system or improving a local area network. Most computer scientists work with people, writing programs in areas such as biology, chemistry, business, economics, publishing, automotive design, or entertainment. Look around you and think about the computer applications you use many times each day: from your browser to an instant messaging program and emails, from word processing to iTunes, cell phones, and iPods.

In addition to our desktop computers, there are the computers we use daily that we do not even think about. For example, the computer in your car that checks your gas mileage several times every second; that examines the wear on your brake pads; that monitors your speed and emissions; that updates your GPS display to keep track of your exact position and plots it on a map on the dashboard display. More and more appliances have computer capabilities as part of their function.

If you go to the doctor and need medical imaging such as a CAT scan or MRI, you are relying on sophisticated computer programs to make and interpret the images of your body. If you look in the cockpit of a newer airplane, you will see that the entire cockpit is nothing more than a bunch of displays with virtual switches on them. In addition, the pilot of the airplane has spent many hours training in a simulated environment controlled by a computer.

There are the computer programs that run behind the scenes that make the world a more orderly place to live. As a student, you probably know that a computer keeps track of your lunch money or credit at the cafeteria, your grade point average, and how much money you owe the college this month. The computers in the library keep track of which books the library owns, who has books checked out, and when they are due back in the library.

If you have ever flown in an airplane, you may not have realized how much the routing and positioning of airplanes rely on computers. Each airplane is tracked by two different radar

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION

systems, plus a transponder in the airplane itself that transmits information to a computer to identify itself, along with its current position and altitude. On the ground a multitude of computers share this information to help route the airplane on a safe and efficient path from takeoff to touchdown. Even after the plane is on the ground, computers continue to track it as it moves from the runway to its gate.

When you arrange for a package to be delivered by UPS, that package is routed and tracked by a very sophisticated computer program. At the UPS worldport in Louisville, Kentucky, every package is automatically photographed, measured, and weighed, and it has the information on its super-barcode analyzed by computers to determine its trajectory along some of the 17,000 conveyor belts. This requires awesome computing power: more data are processed here every 30 minutes than in an entire day of trading on the New York Stock Exchange. Eventually every package slides down a chute and is placed into a bag or an air-freight container. And before dawn it is off again to complete its journey in another aircraft or in one of a fleet of waiting trucks [Eco06].

In large cities all the stoplights are controlled by computers. Sensors in the road monitor traffic conditions. Large simulations are used to determine how to keep traffic flowing as quickly as possible. When high traffic is detected in an area, the computer adjusts the timing of the stoplights to increase the flow of traffic [How97].

Computers are providing key help on the forefront of biological research with proteins. Before proteins can carry out their important functions as biology's workhorses, they assemble themselves and undergo the process of protein "folding." While critical and fundamental to virtually all of biology, in many ways the process remains a mystery. Moreover, when proteins do not fold correctly, there can be serious consequences—from Alzheimer's disease to mad cow, Huntington's and Parkinson's disease, as well as many cancers and cancer-related syndromes. The folding@home project is using computers throughout the world to try to understand how protein folding works. For example, the best-known gene that helps to fight cancer is called p53. Roughly half of all known cancers result from mutations in this gene. One of the first published results from the folding@home project involves an investigation into p53 folding. Researchers at Stanford say "We predict how p53 folds and in doing so, we can predict which amino acid mutations would be relevant. When compared with experiments, our predictions have appeared to agree with the experiment and give a new interpretation to existing data." [fold]

In the war on terror, computers are not only busy inside government agencies analyzing millions of emails from suspected terrorists but also working on ways to respond to every possible kind of attack. For example, suppose that the smallpox virus were released into the population of a city like Portland, Oregon. What would be the best response to prevent an epidemic? Should the government institute a policy of mass vaccination? Should only

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION

exposed individuals and their circle of contacts be quarantined and/or vaccinated? Should major centers where people congregate, like schools and malls, be closed? These questions can best be answered by using massive computer simulation programs that can anticipate the movements of individuals in a large city, along with various strategies for reducing the spread of a disease [BES05].

Countless motion pictures today are produced with various kinds of computer-generated special effects. Obviously there are the computer-animated movies such as *Toy Story*, *Cars*, and *Shrek*, but the line is continually blurring between live action and computer-generated films. For example, the vast battles between the Orcs and the humans in the *Lord of the Rings* movies were completely computer-generated, including the actions of each individual battle participant. In the traditionally animated movie *The Lion King*, the wildebeest stampede was computer-generated. For his role in *The Polar Express*, Tom Hanks wore a special suit with sensors on it so that his body motions were continuously monitored. After the live action part of the film was completed, a new skin was computer-generated and put on the body that followed the live movements of the actor.

1.3.2 Why Computer Science Is Important

In one sense, studying computer science is important because it gives you a better understanding of how the technological world around you works. After reading this book, you may have a new appreciation for the applications you use every day. Hopefully, you will even aspire to write your own improved version of some application.

In another sense, computer science helps to prepare you for your future. Many of the interesting jobs are going to be at the intersection of computer science and some other domain. Speaking at the Microsoft Research Faculty summit, Bill Gates made this observation: "The nature of these jobs is not just closing the door and coding. The greatest missing skill is somebody who's good at understanding engineering and bridges that to working with customers and marketing." Recently a manager at a successful software company said, "I sometimes turn away applicants who have perfect scores on the standardized tests." The reason he said this was that these applicants may be technically perfect, but they lack the ability to communicate with people and solve real-world problems.

Most importantly, a first course in computer science will help you in the following ways:

- You will be able to apply new problem-solving skills.
- You will learn to apply logic.
- You will learn about process (a series of actions or steps taken in order to achieve a specific outcome).

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION

- You will understand and apply abstraction.
- You will learn to think and communicate more clearly.

1.4 Problem-Solving Strategies

Problem solving happens on three different levels:

- **Strategy:** A high-level idea for finding a solution.
- **Tactics:** Methods or patterns that work in many different settings.
- **Tools:** Specific tricks and techniques that are used in specific situations.

Paul Zeitz [Zei99] provides us with a helpful analogy for illustrating the three different levels of problem solving:

> You are standing at the base of a mountain, hoping to climb to the summit. Your first strategy may be to take several small trips to various easier peaks nearby, so as to observe the target mountain from different angles. After this, you may consider a somewhat more focused strategy, perhaps to try climbing the mountain via a particular ridge. Now the tactical considerations begin: how to actually achieve the chosen strategy. For example, suppose that our strategy suggests climbing the south ridge of the peak, but there are snowfields and rivers in our path. Different tactics are needed to negotiate each of these obstacles. For the snowfield, our tactic may be to travel early in the morning while the snow is hard. For the river, our tactic may be scouting the banks for the safest crossing. Finally, move into the most tightly focused level, that of tools: specific techniques to accomplish specialized tasks. For example, to cross the snowfield we may set up a particular system of ropes for safety and walk with ice axes. The river crossing may require the party to strip from the waist down and hold hands for balance. These are all tools. They are very specific. You would never summarize, "To climb the mountain we had to take our pants off and hold hands," because it was a minor—though essential—component of the entire climb. On the other hand, strategic and sometimes tactical ideas are often described in your summary: "We decided to reach the summit via the south ridge and had to cross a difficult snowfield and a dangerous river to get to the ridge."

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION

As you progress through this book, you will encounter several different problem-solving strategies. In addition, you will see that computer science uses many different problem-solving tactics. In particular you will learn to recognize patterns in the problems you solve that lead to patterns in the programs you write. Finally, as you use the Python programming language you will learn about the tools that Python provides to write your solution as a program.

A simple example will illustrate some of what we are talking about. The question is as follows: "A class has 12 students. At the beginning of class each student shakes hands with each of the other students. How many handshakes take place?"

Your first instinct might be to simply say that since each person must shake hands with 11 other people the answer is $12 \cdot 11 = 132$ handshakes, but you would be wrong. To help you make progress toward the correct answer, you can employ a strategy called **simplification**. Simplification is a strategy that reduces a problem to a trivial size.

Let's assume that instead of 12 people there is only one person in the classroom. When there is only a single person, no handshakes will take place. But what happens when a second person enters the classroom? Upon entering the room, the second person must shake hands with the first (and only other) person in the room for one handshake. Now suppose a third person enters the classroom. The third person must shake hands with the first two, making a total of $2 + 1 + 0 = 3$ handshakes. The fourth person who enters the room must shake hands with the three people already in the room, so our total handshake count is now $3+2+1+0 = 6$. By this time you should see a pattern of **generalization**—a technique that enables you to go from some specific examples to a solution that can be implemented as a program.

In our handshaking problem the pattern is telling us that the Nth person to enter the classroom shakes hands with $N-1$ other people, and the total number of handshakes is the sum $1 + 2 + 3 + \ldots N - 1$. At this point we might simply write a computer program that adds up the numbers from 1 to $N-1$ for us. Adding numbers is something that computers are particularly good at. However, we will also point out that there is a general solution for this problem for adding a sequence of numbers:

$$sum = \frac{n \cdot (n+1)}{2}$$

For our handshake problem, we need to add up the numbers from 1 to 1 less than the number of students. Given that there are 12 students, $n = 11$. Plugging 11 into the formula gives us:

$$\frac{11 \cdot 12}{2} = 66$$

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION

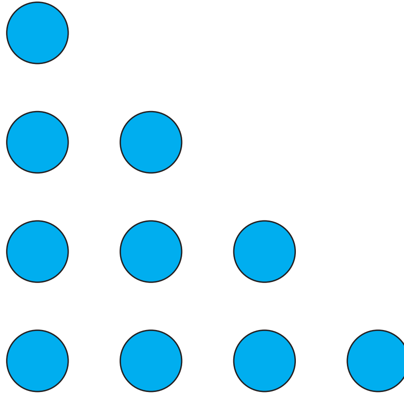

Figure 1.1 Representing the sum of the numbers from 1 to N graphically

You can verify this result yourself by simply adding the numbers from 1 to 11.

In fact we can *prove* that the formula gives us the correct answer by using **representation**, another important strategy that will solve our problem. Proving that $\sum_{i=1}^{n} i = \frac{n \cdot (n+1)}{2}$ is true for all values of n using mathematical induction would be a daunting task for most people. However, let's visualize the problem of adding up the numbers from 1 to N as shown in Figure 1.1.

In this representation of the problem, we show each of the numbers we want to add as a row of circles, thus representing the addition of $1 + 2 + 3 + 4$. Now we could just count the circles to get our answer, but that is not very interesting and does not prove anything. The interesting part comes in Figure 1.2, where we have taken all four rows of dots, duplicated them, and flipped them diagonally. The dots now form a rectangle that is 4 rows high and 5 columns wide. It is now easy to see that the total number of dots is just $4 \cdot 5 = 20$. But we have twice as many dots as we started with, so the number of dots we started with is $20 \div 2 = 10$.

If you generalize our example a little bit, it is easy to see that this graphical trick works no matter how many rows of dots we use. Therefore, we have shown a proof for an interesting mathematical sequence, but because we chose a good representation for the problem we have not had to do anything more complicated than simple multiplication and division.

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION

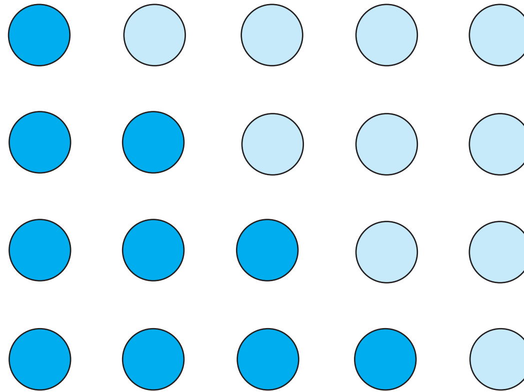

Figure 1.2 The sum of the numbers 1 to N is $\frac{n \cdot (n+1)}{2}$

1.5 Python Overview

In this book the language you will use to write your computer programs is called **Python**. Why did we choose Python instead of a language like C++ or Java? The answer is simple: We want you to focus on learning the problem-solving strategies and techniques that a computer scientist uses. **Programming languages** are tools and Python is a good beginning tool. Languages like Java and C++ are fine tools as well, but they require you to keep track of many more details and they are harder to learn than Python.

The best way to learn Python is to try it out—so let's get started. The first thing we are going to do is start the Python interpreter. Depending on your operating system, there are any number of ways to do this. For example, you might start a program called IDLE—named after Eric Idle of Monty Python fame. Or you might just type `Python` at the command prompt. No matter how you start it, you will know you are successful when you see a window such as the one shown in Figure 1.3. In this case, we have started the Python interpreter from a terminal window on a MacBook Pro. For detailed instructions on installing and starting Python on your system, refer to Appendix A.

As you progress through this chapter, you will see that example programs are in boxes called **listings**, and commands that you can type interactively at the Python shell are in boxes called **sessions**. Whenever you see a session box, we strongly encourage you to try the session for yourself. Also, once you have typed in the example we have shown, feel free to try some variations in order to find out for yourself what works and what does not.

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION

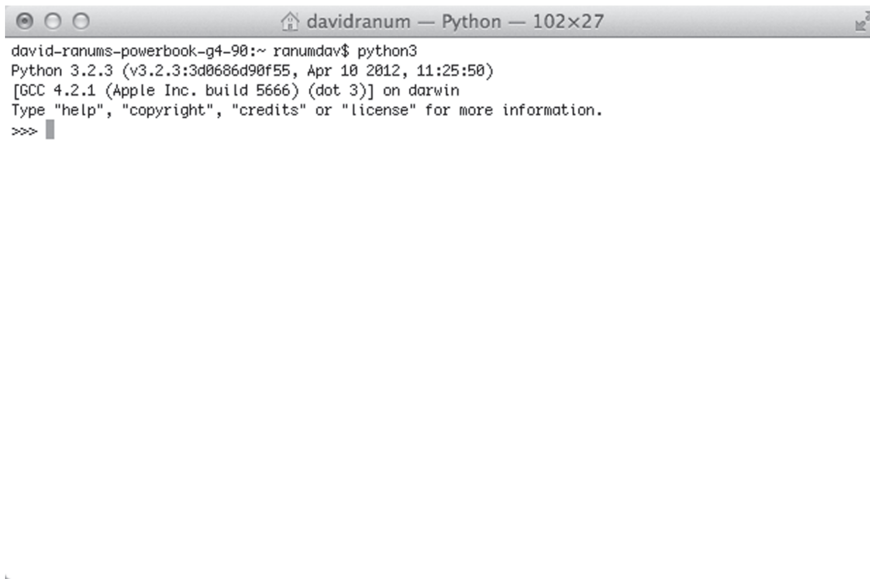

Figure 1.3 The Python shell

As we begin to explore Python, we will answer three important questions you should ask about any programming language:

- What are the primitive elements?
- How can we combine the primitive elements?
- How can we create our own abstractions?

1.5.1 Primitive Elements

At the deepest level, the one primitive element in Python is the **object**. In fact, everything in Python is an object, and you will read this refrain often in this book. By now you are probably wondering what we mean by *object*. After all, if you look around you will see many objects: this book, pencils, pens, your chair, a computer. What do these items have to do with Python? Like you, Python thinks of the things in its world as objects. Python even considers numbers to be objects—an idea that may be a bit confusing to you as you probably don't think of numbers as objects. But Python does, and we'll see why this is important shortly.

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION

Python classifies the different kinds of objects in its world into **types**. Some of the easiest types to work with are numbers. Python knows about several different types of numbers:

- Integer numbers
- Floating point numbers
- Complex numbers

Integer Numbers

Integers are the whole numbers that you learned about in math class. We will introduce more of Python's primitive types as we progress through this chapter. But before we move on let's look at Python's integers in more detail. We can already do a lot with Python just using integers. For starters, we can use the Python shell we started a few moments ago as a calculator. Let's try a few mathematical expressions. Type in the following examples using the Python interpreter. After you have typed in an expression, press the return key to see the result.

```
>>> 2+2
4
>>> 100-75
25
>>> 7*9
63
>>> 14//2
7
>>> 15//2
7
>>> 15 % 2
1
```

Session 1.1 Simple integer math

The examples in Session 1.1 illustrate some very important Python concepts that you should become familiar with as soon as possible. The first concept is Python's evaluation loop.

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION

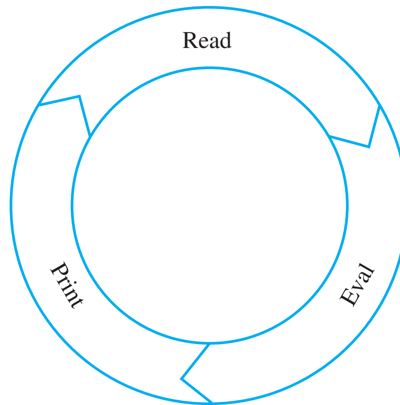

Figure 1.4 The Read–Eval–Print loop in Python

At a high level, the Python interpreter is really very simple. It does three things over and over again: (1) read, (2) evaluate, and (3) print. These are illustrated in Figure 1.4.

First, Python *reads* one line of input. In the first example, Python reads 2 + 2, then it *evaluates* the expression 2 + 2 and determines that the answer is 4. Finally, Python *prints* the resulting value of 4. After displaying the result, Python prints the characters `>>>` to show you that it is waiting to read another expression. The three characters `>>>` are called the Python **prompt**.

In general, a Python **expression** is a combination of operators and operands. When we evaluate an expression, we get a result. In the examples in our Python session, the operators are familiar mathematical operators '+', '−', '*', and '//'. You may be more used to × and ÷ for multiplication and division, but you will not find those symbols on a standard keyboard, so Python, and almost all other programming languages use "*" and "/".

One thing that may surprise you in the example is the result of the expression `15//2`. Of course, we all know that 15 divided by 2 is really `7.5`. However, because both operands are

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION

integer objects and // is the integer division operator, Python produces an integer object as a result. Integer division works like the division you learned when you were young. 15 divided by 2 equals 7, remainder 1. You can find out the remainder part of the result using the modulo operator (%). For example, `15 % 7` evaluates to the remainder value of 1.

Integer division is really useful in some cases, but it can also trip you up. What if you want to divide 15 by 2 and get `7.5` as the answer? In order to get the result as a **floating point** number you must use the floating point division operator '/'.

Exercises

1.1 Find the sum of the numbers 8, 9, and 10.

1.2 Find the product of the numbers 8, 9, and 10.

1.3 Compute the number of seconds in a year.

1.4 Compute the number of inches in 1 mile.

1.5 Compute the number of 2 ft square tiles to cover the floor of a 10 by 12 ft room.

1.6 Compute the number of handshakes required for each person in your class to shake hands with every other person exactly one time.

1.7 Find the average age of five people around you using integer division. Doublecheck your answer.

Floating-Point Numbers

Floating-point numbers are Python's approximation of what you called real numbers in math class. We say that floating-point numbers are an approximation because unlike real numbers, floating-point numbers cannot have an infinite number of digits following the decimal point. In Python you can tell the difference between a floating-point number and an integer because a floating-point number has a decimal point. Session 1.2 presents some examples of math using floating-point numbers.

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION

```
>>> 2.0 + 2.0
4.0
>>> 2 + 2.0
4.0
>>> 15 / 2
7.5
>>> 2.0 ** 50
1125899906842624.0
>>> 2.5 ** 25
8881784197.0012531
>>> 2.0 ** 500
3.2733906078961419e+150
>>> 1.33e+5 + 1.0
133001.0
```

Session 1.2 Floating-point math

Notice that we have added something new in this example: the `**` symbol, which is called the exponentiation operator. So `2.0 ** 50` is really two to the fiftieth power. You should also notice that when the result of a floating-point operation gets really big, Python uses scientific notation to express the results. The Python number `3.273e+150` really means 3.273 times 10 to the 150th power, or 3273 followed by 147 zeros! A very big number indeed. Notice also that you can use floating-point numbers in scientific notation as part of a Python expression.

Exercises

1.8 Find the average age of five people around you using floating-point division. Double-check your answer.

1.9 Find the volume of a sphere with a radius of 1 using the formula $4/3\pi r^3$.

1.10 Compute 1/3 of 15. Did you get the right answer?

1.11 The Andromeda galaxy is 2.9 million light years away. There are 5.878×10^{12} miles per light year. How many miles away is the Andromeda galaxy?

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION

1.12 How many years would it take to travel to the Andromeda galaxy at 65 miles per hour?

Although 3.273e + 150 is a good approximation, we know that there are not really 147 zeros in the result. One of the disadvantages of using scientific notation is that you lose some *precision* in your result. If you want to get very exact results, integers allow us to do calculations to unlimited precision. Session 1.3 shows the real value of `2 ** 500` using integers.

```
>>> 2 ** 500
32733906078961418700131896968275991522166420460430647894832913680961
33796404674554883270092325904157150886684127560071009217256545885393
053328527589376
>>>
```

Session 1.3 The use of integers to obtain very precise answers for large numbers

Exercises

1.13 Compute the factorial of 13.

1.14 Compute 2 to the 120th power.

1.15 If the universe is 15 billion years old, how many seconds old is it?

1.16 How many handshakes would it take for each person in Chicago to shake hands with every other person?

Complex Numbers

The final primitive numeric type in Python is the **complex number**. As you may remember, complex numbers have two parts to them, a real part and an imaginary part. In Python a complex number is displayed as *real* + *imaginary*`j`. For example, $5.0 + 3j$ has a real part of 5.0 and an imaginary part of 3. Although we mention complex numbers here to give you a complete list of the numeric primitives, we will not go into any additional details at this point.

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION

Summary of Numeric Types

What happens when we mix integers and floating-point numbers? Let's look at the examples shown in Session 1.4 to find out.

```
>>> 100 * 3.4
340.0
>>> 100000000000000000000000000 * 3.4
3.4000000000000003e+26
>>> 10000000 * 1000000
10000000000000
>>> 1000000000 / 1000000000
1
>>> 1000 // 10.0
100.0
>>> 1000 / 10.2
98.039215686274517
>>> 1000 // 10.2
98.0
>>> 5 + 4+3j
(9+3j)
```

Session 1.4 Mixing integers, long integers, floats, and complex numbers

When mixing different types of numbers, you can figure out what the result will be converted to by applying the following rules:

1. If either argument is a complex number, the other is converted to complex.
2. If either argument is a floating-point number, the other is converted to floating point.
3. For all other arguments, both must be plain integers and no conversion is needed.

Notice that when using floating-point numbers with the integer division operator the result is a floating-point number with the fractional part truncated. You can also tell Python to explicitly convert a number to either an integer or floating-point number by using the

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION

`int` or `float` functions. For example, `float(5)` will convert the integer 5 to the floating-point number 5.0. When converting floating-point numbers to integers, Python always truncates the fractional part of the number. For example, `int(3.99999)` will convert the floating-point number 3.99999 to the integer 3.

In summary, we have seen that Python supports several different types of primitive objects in the number family: integers for ordinary simple math; or, when precision is required or when dealing with very large numbers; floating-point numbers, for working with scientific applications or accounting applications where we need to keep track of dollars and cents. We have seen that Python can be used to make simple numerical calculations. However, at this point, Python is nothing more than a calculator. In the next section we will add some additional Python primitives that will give us a lot more power.

1.5.2 Naming Objects

Very often we have an object that we would like to remember. Python allows us to **name** objects so that we can refer to them later. For example, we might want to use the name *pi* rather than the value 3.14159 in a mathematical expression. We might also want to give a name to a value that we are going to use over and over again rather than recalculating it each time.

In Python we can name objects using an **assignment statement**. A statement is like an expression except that it does not produce a value for the read–eval–print loop to print. An assignment statement has three parts: (1) the left-hand side, (2) the right-hand side, and (3) the assignment operator (=). The left side contains the name we are assigning to and the right side can be any Python expression.

```
name = python expression
```

When the Python interpreter evaluates an assignment statement, it first evaluates the expression that it finds on the right-hand side of the equals sign. Once the right-hand side expression has been evaluated, the resulting object is referred to using the name found on the left side of the equals sign. In computer science, we call these names **variables**. More formally, we define a variable to be a named reference to a data object. In other words, a variable is simply a name that allows us to locate a Python object.

Suppose we want to calculate the volume of a cylinder where the radius of the base is 8 cm and the height is 16 cm. We will use the formula $volume = area\ of\ base * height$. Rather than calculate everything in one big expression, we will divide the work into several assignment statements. First, we will name the numeric objects “pi,” “radius,” and “height.” Next,

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION

we will use the named objects to calculate the area of the base and finally the volume of the cylinder. Session 1.5 shows how we use this sequence of assignment statements and Python arithmetic to solve our problem.

```
>>> pi = 3.14159
>>> radius = 8.0
>>> height = 16
>>> baseArea = pi * radius ** 2
>>> cylinderVolume = baseArea * height
>>> baseArea
201.06175999999999
>>> cylinderVolume
3216.9881599999999
>>>
```

Session 1.5 Calculating the volume of a cylinder with assignment statements

After studying Session 1.5, you may have some questions:

- How is the use of the equals sign in Python different from what you learned in math class?
- If you change the value for `baseArea` will `cylinderVolume` automatically change?
- Why doesn't Python print out the value of `pi` after the first assignment statement?
- What names are legal in Python?

Let's look at these questions one at a time. The equals sign in Python is very different from what you learned in math class. In fact, you should think of it not in terms of equality but rather as the assignment operator, which has the job of associating a name with an object. Figure 1.5 illustrates how names are associated with objects in Python. All the names and objects in this figure come from Session 1.5. The relationships between names and the objects they reference are indicated by the arrows between them.

Another way of thinking about assignment is to imagine that an assignment statement is like taking a sticky label with a name written on it and attaching it to an object. You know that you can put more than one sticky label on an object in the real world, and the

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION

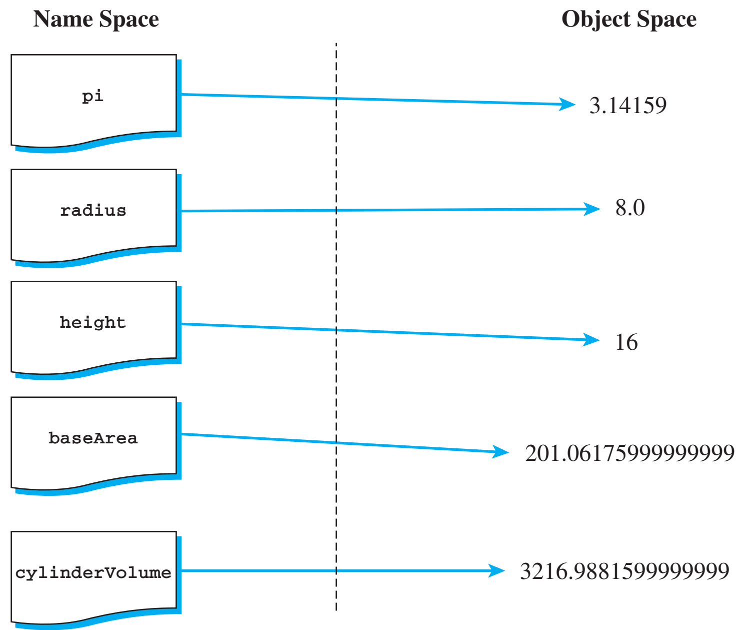

Figure 1.5 A reference diagram illustrating simple assignment in Python

analogy holds true in the Python world as well. A Python object can have more than one name. For example, suppose you made the following additional assignment $x = 8.0$. After executing that statement, you could add another label called `x` with another arrow pointing at the object 8.0, as shown in Figure 1.6.

Variables can take the place of the actual object in a Python expression. When Python evaluates the expression `pi * radius ** 2`, it first looks up `pi` and `radius` to see what objects they refer to and then substitutes the values into the expression. The expression thus becomes `3.14159 * 8.0 ** 2`. Next, Python evaluates `8.0 ** 2` to get an intermediate result of `64.0`. Python then evaluates `3.14159 * 64.0` to get the value `201.06176`. After the right-hand side of the expression is evaluated, Python assigns the name `baseArea` to `201.06176`.

Let's look at one more example using assignment. Consider the Python statements in Session 1.6.

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION

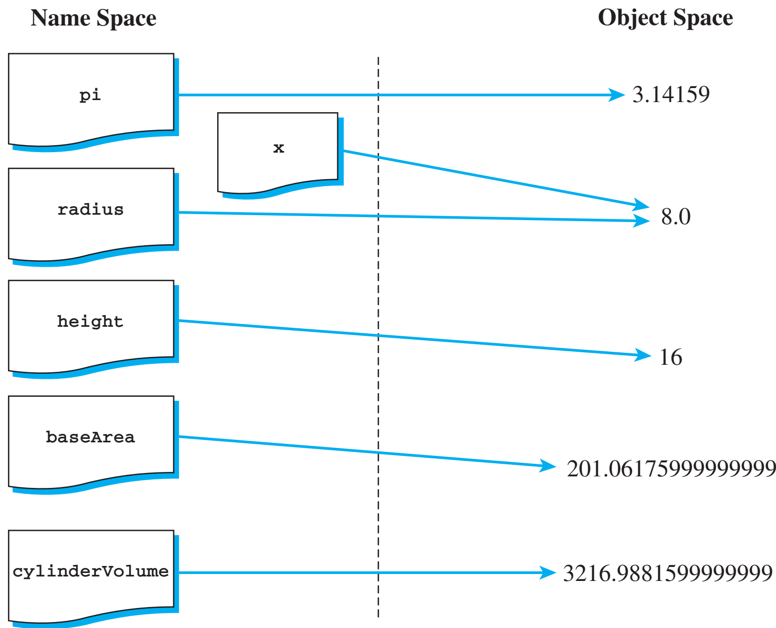

Figure 1.6 Reference diagram after x = 8.0

```
>>> a = 10
>>> b = 20
>>> a = b
```

Session 1.6 Python evaluates assignment statements in sequence

After these three assignment statements, what object does `a` refer to? As you think about this question it is very important to remember that Python evaluates the statements from top to bottom, one after another. Let's rephrase what is going on in the order that Python performs its work.

1. Assign the name `a` to the integer object 10.
2. Assign the name `b` to the integer object 20.
3. Find the object named `b`, then assign the name `a` to that object.

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION

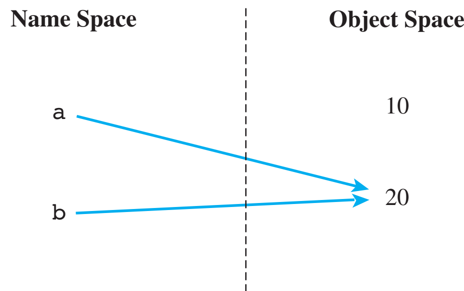

Figure 1.7 Result of assignment `a = b`

The answer to our question is that `a` now refers to the object 20. This is shown in Figure 1.7. In addition, since we have not changed what `b` refers to since the original assignment, `b` continues to refer to the object 20. If you are confused by this example, try to draw the reference diagram yourself one step at a time.

Now that you understand more about the assignment operator you will understand that attaching the name `baseArea` to a different object will have no impact on the name `cylinderVolume`. To make the new value for `baseArea` change the value of `cylinderVolume`, you will need to ask Python to execute the statement `cylinderVolume = baseArea * height`.

Since assignment is a statement rather than an expression, it does not return a value. This means that there is nothing for the read–eval–print loop to print out. That is why you do not see any output following the assignment statements in Sessions 1.5 and 1.6.

A name on a line all by itself is a very simple Python expression. Notice that just typing a name causes Python to find the value for the object and return it as the result of the expression. You can see examples of this in Session 1.5.

There are several important rules to remember about naming things in Python. Names can include any letter, number, or an _ (underscore). Names must start with either a letter or an _. Python is case sensitive, which means that the names `baseArea`, `basearea`, and `BaseArea` are all different. Some names are reserved by Python for its own use. These are called **keywords** and correspond to important Python capabilities that you will learn about. Table 1.1 shows you all of Python's reserved keywords.

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION

and	continue	else	for	import	not	raise
assert	def	except	from	in	or	return
break	del	exec	global	is	pass	try
class	elif	finally	if	lambda	print	while

Table 1.1 Python's reserved words

Exercises

1.17 Given the following Python statements:

```
a = 79
b = a
a = 89
```

(a) Draw a reference diagram to show the labels and objects after the first two statements.

(b) Draw a reference diagram to show the labels and objects after the last statement.

1.18 Which of the following are legal variable names:

(a) `_abc123`

(b) `123abc`

(c) `abc123_`

(d) `_123`

1.19 Consider the following statements:

```
a = 10
b = 20
c = a * b
d = a + b
```

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION

Draw a reference diagram to show all the objects and names after evaluating these statements

1.20 What are the values of `a` and `b` after Python evaluates each of the following four statements?

```
a = 10
b = 20
a = b
b = 15
```

1.21 Consider the following statements:

```
idx = 0
idx = idx + 1
idx = idx + 2
```

What is the value of `idx` after Python evaluates each of the three statements?

1.5.3 Abstraction

Abstraction is defined as a concept or idea not associated with any specific instance. For example, you can think of mathematical functions on your calculator such as square root, sine, and cosine as abstractions. These functions can calculate a value for *any* number, but the method of calculating a square root is independent of the particular number. In that way we can think of a function that calculates a square root as being the general idea. The square root function works equally well for all numbers because it is a general function. We do not have a special square root function for each possible number, just one function that works for all numbers.

One way of thinking about functions is as a "black box." You send information into the black box on one side and new information comes out on the other side. You don't know exactly what goes on inside the box but you do know the behavior that the box should exhibit. Figure 1.8 illustrates this concept for the square root function.

The Python language contains many such abstractions. Many of the new things we will see in Python from here on are in fact abstractions built using the Python primitives we have already talked about or will talk about in the first few chapters of this book. In other words, much of Python is written in Python.

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION

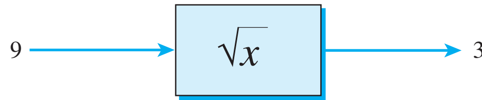

Figure 1.8 A black box view of the square root function

The `turtle` Module

Many of the additional parts of Python functionality are found in **modules**—an optional part of Python that implements an abstraction that is designed to make programming easier. In order to get the power of a module, you have to tell Python to load the module you want. The statement you need to use to load a module is `import` *modulename*.

When you `import` a module, an object is created inside Python. That object has the type *module* and has a name attached to it that matches the name you used on the import line. Every object in Python has three important characteristics: (1) an identity, (2) a type, and (3) a value. In addition, some Python objects have special values called attributes, and some Python objects also have `methods` that allow us to ask the object to do something interesting. Let's look at a simple example before we go any further.

The example we will use is the `turtle` module. The `turtle` module provides us with a simple graphics programming tool known as a turtle. Very simply, a turtle is an object that we can control. A turtle can move forward or backward, and it can turn in any direction. When a turtle moves, it draws a line if its tail is down. A turtle is a Python object that has both attributes and methods. Some of the attributes of a turtle are shown in Table 1.2.

position	The coordinates of the turtle on the screen
heading	The direction the turtle is facing
color	The color of the turtle
tail position	The turtle's tail can be up or down

Table 1.2 Turtle attributes

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION

Name	Parameter(s)	Description
`Turtle`	None	Creates and returns a new turtle object
`forward`	Distance	Moves the turtle forward
`backward`	Distance	Moves the turle backward
`right`	Angle	Turns the turtle clockwise
`left`	Angle	Turns the turtle counterclockwise
`up`	None	Picks up the turtle's tail
`down`	None	Puts down the turtle's tail
`color`	Color name	Changes the color of the turtle's tail
`fillcolor`	Color name	Changes the color that the turtle will use to fill a polygon
`heading`	None	Returns the current heading
`position`	None	Returns the current position
`goto`	`x, y`	Moves the turtle to position `x, y`
`begin_fill`	None	Remembers the starting point for a filled polygon
`end_fill`	None	Closes the polygon and fills it with the current fill color
`dot`	None	Leaves a dot at the current position

Table 1.3 Summary of simple turtle methods

The methods of the turtle are summarized in Table 1.3. To start, we will just concern ourselves with a few of them. For example, we can tell the turtle to go `forward`, `backward`, turn `left`, turn `right`, or ask for its `position`. The turtle has a tail that can be `up` or `down`. When the tail is down and the turtle moves, it draws a line. If the tail is up and the turtle moves, nothing is drawn. Session 1.7 shows a Python session where we create a `turtle` object and try out some of the turtle's capabilities.

Let's look at this example line by line. In line 1 we use an import statement to load the `turtle` module. In line 2 we ask Python to evaluate the name `turtle`. Python rewards us by telling us the identity of the object that `turtle` is assigned to. If you look at the identity, you will see it is telling the location of the source code for the module. This location will vary according to the kind of computer you are using. If we were really adventurous, we could go there and look at the Python methods that are stored in the `turtle.py` file.

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION

```
>>> import turtle
>>> turtle
<module 'turtle' from
'/Library/Frameworks/Python.framework/Versions/3.2/lib
/python3.2/turtle.py'>
>>> gertrude = turtle.Turtle()
>>> gertrude
<turtle.Turtle object at 0x101393950>
>>> gertrude.forward(100)
>>> gertrude.right(90)
>>> gertrude.forward(50)
>>> gertrude.position()
(100.00,-50.00)
>>> gertrude.heading()
270.0
>>>
```

Session 1.7 Using the `turtle` module

Once we have the `turtle` module loaded, we will start to use the methods in the module to do something interesting. In line 6 we have an assignment statement, in which we are making a new `Turtle` object and giving it the name `gertrude`.

Before we go any further with this example, we need to explain line 6 in more detail. In particular the expression `turtle.Turtle()` contains a new operator—the `dot (.)`. The **dot operator** tells Python to look up the name right in front of the dot and return the object it names. This process is referred to as **dereferencing**. In this case the dereferencing operation allows Python to find the `turtle` module. Once we have the module, Python continues looking for the name to the right of the dot. In this case Python looks for the name `Turtle`. A good way to think of this is that Turtle is "inside" the `turtle` module. So the first `.` gets us to the `turtle` module and then inside the `turtle` module we find the object named `Turtle`.

We can then see that `turtle.Turtle()` is a method that creates a new object. The type of the new object is `Turtle`. A method that creates a new object is called a **constructor**. New turtles that we construct are called **instances** of the type `Turtle`.

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION

If you are typing this session interactively as you are reading, you will see that when a turtle is created a new window appears on the screen. The triangle in the middle of the screen is the turtle. When a new instance of `Turtle` is first created, it is at position (0, 0) in the middle of the window. The turtle's initial heading is 0 degrees, or straight to the right. The color attribute for the new turtle is black. As you move the turtle around, it remembers its latest position, what direction it is facing, and whether its tail is up or down.

The next two lines simply demonstrate that when you evaluate a name that corresponds to a more complicated object you get back Python's representation of that object. Unlike a number where the representation is self-evident, a turtle's representation gives you a unique identification for the object. In this case we see that `gertrude` is `<turtle.Turtle object at 0x101393950>`. To be even more specific, the result is telling us that the type of `gertrude` is `turtle.Turtle`. Furthermore, the turtle can be found at location `0x101393950` in the computer's memory.

Now that we have our new turtle object, named `gertrude`, the next three lines instruct `gertrude` to do some drawing. As you might guess, the line `gertrude.forward(100)` causes the turtle to move forward 100 units. Once again, the dot notation is very important to understanding how Python interprets the statement. First, the dot tells Python to dereference the name `gertrude`. When Python finds the object, it evaluates the method `forward` that is "inside" the turtle. The forward method knows that it needs to move forward 100 units because we pass it a parameter of 100. Just as mathematical functions like $\cos(20)$ or $\sqrt{20}$ take parameters, Python functions and methods may also accept parameters.

Functions are abstractions of generalized behaviors. **Parameters** are the way we tell the function *specifically* what it should do. In this case we want `gertrude` to move forward 100 units, then turn right 90 degrees, then move forward again, but this time by only 50 units. If you try to run this example on your own, your window should look just like Figure 1.9.

The last four lines of Session 1.7 show that we can also use methods to ask the turtle for information about itself. First, we ask `gertrude` to tell us where it is with the method call `gertrude.position()`. `gertrude` replies that it is at (100.0, −50.0). What is in fact happening is that the `position()` method returns the value (100.0, −50.0) and the print part of the read–eval–print loop prints out that value. All functions can return values, and so they can be included in Python expressions. The fact that the `position` method returns a value is no different than the cosine function returning a value. Similarly the heading method tells us that `gertrude` is currently facing 270 degrees.

Notice that in the world of our turtle, the coordinates (0, 0) are in the center of the window. The x coordinate grows in a positive direction as the turtle moves toward the right. If the turtle moves toward the left side of the window, the x coordinates get smaller and are

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION

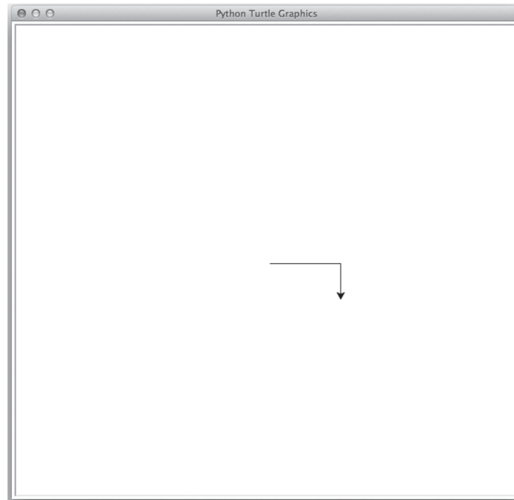

Figure 1.9 Using methods to control a turtle

negative to the left of the middle of the window. Similarly, the y coordinate grows as the turtle moves toward the top of the window and gets smaller as the turtle moves toward the bottom. One pixel on your computer screen corresponds to 1 unit of turtle movement. If the turtle's heading is 0 degrees, it is facing to the right; 90 degrees is up, 180 degrees is to the left, and 270 degrees is down.

The turtle objects found in the `turtle` module have many other methods. Table 1.3 shows just a few of them. We'll introduce additional methods as we have need for them in future examples.

Exercises

1.22 Create a turtle called `sven`. Now tell `sven` to go forward 10. What is `sven`'s position now?

1.23 Create a turtle called `ole` and tell `ole` to turn right 45 degrees and go forward 50. Notice that you now have two turtles in the same window.

1.24 On a sheet of graph paper sketch out a simple line drawing of something. Using the turtle methods in Table 1.3, recreate your line drawing.

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION

Writing Your Own Functions

We are not limited to the functions that the authors of Python have given us. We, too, can write our own functions to add our own abstractions to the Python language. In fact, functions are just another kind of Python object with a couple of special capabilities. We `define` a function in Python using the *def* statement. Listing 1.1 shows a general template for using the *def* statement to define a function.

```
1  def functionName(param1,param2,...):
2      statement1
3      statement2
4      ...
```

Listing 1.1 A template for function definition

The *def* statement begins by giving the function a name. Next we specify any parameters we want our function to accept. In Python we can have zero or more parameters for any function we write. All the parameters must be listed inside the parentheses that follow the function name. Next we have a colon character, which tells the Python interpreter that the sequence of indented statements that follow are all part of the function. Python knows to stop reading lines for the *def* statement when it encounters a line that is not indented. We call a group of Python statements that are indented at the same level a **block**.

Now let's try to write a real function. Suppose that we are working on a graphics program that requires us to draw a lot of squares. Telling the turtle how to draw a square each time we need one is tiresome and repetitious. In fact, a square is an example of an abstraction. We know that a square is a geometric shape that has four sides of equal length, and four 90-degree corners. What we don't know is how long the sides are for any particular square.

Our goal is to write a function that can use any turtle to draw a square of any size. That statement suggests that we need two pieces of information to solve our problem—namely, the turtle that will draw the square and the size of the square. These pieces of information will become the parameters to our function.

The next step is to use the parameters along with the built-in methods of a turtle. We can draw a square by moving the turtle forward and turning right by 90 degrees four times in a row. Listing 1.2 shows a complete Python function for drawing a square with a turtle.

Notice that the statements in the function are the same turtle commands that we typed in interactively when we first started using the turtle. When the commands are inside the function, they are just grouped together so that we can have all the commands run as if we had typed them one after another. This is one of the great powers of writing a function.

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION

```
1  def drawSquare(myTurtle,sideLength):
2      myTurtle.forward(sideLength)
3      myTurtle.right(90)    # side 1
4      myTurtle.forward(sideLength)
5      myTurtle.right(90)    # side 2
6      myTurtle.forward(sideLength)
7      myTurtle.right(90)    # side 3
8      myTurtle.forward(sideLength)
9      myTurtle.right(90)    # side 4
```

Listing 1.2 A function to draw a square using a turtle

It is also important to remember that Python will do the commands in exactly the order they are typed in the function.

One final thing to note about the code in Listing 1.2 is the # character as seen on Lines 3, 5, 7, and 9. This is known as the **comment marker**. Any text following the comment marker is ignored by the Python interpreter. Comments allow the programmer to place descriptive documentation into Python code that will not impact the execution of the program.

Type the `drawSquare` function into a text file exactly as shown here. After you type in the function, save it to a file called `ds.py`. You have just created your first module! In the same way that `turtle` is a module created by other Python programmers, you have created a module called `ds`. We can now use the `drawSquare` function, as shown in Session 1.8. After running the commands in this session, you should have an image that looks like Figure 1.10.

```
>>> from ds import *
>>> import turtle
>>> t = turtle.Turtle()
>>> drawSquare(t,150)
>>>
```

Session 1.8 A Python session to demonstrate calling a function

There is a lot going on in this simple example, but there are only a couple of new things that you have not seen before. First, we import the `turtle` and `ds` modules. We then make a new `turtle` object named `t`, and next we call `drawSquare`. Python knows that it should treat the object `drawSquare` like a function call because of the parentheses after the name. The parentheses are operators that tell Python to treat the object as a function. You can even show yourself that `drawSquare` is a plain old object by simply typing `drawSquare`

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION

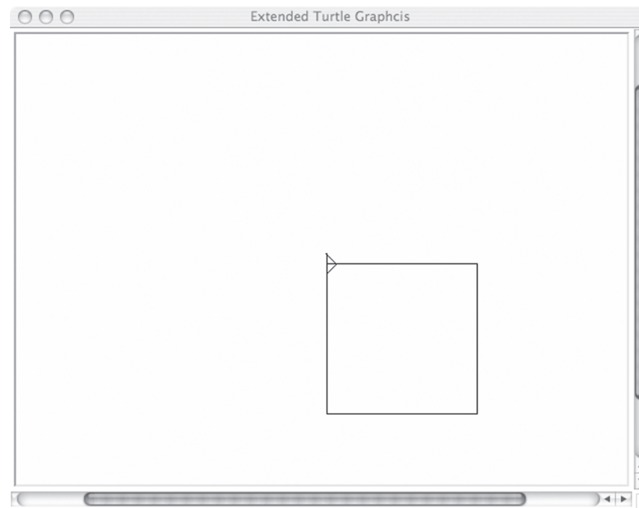

Figure 1.10 The result of the statements in Session 1.8

without the parentheses after the name. If you do so, Python will tell you something like `<function drawSquare at 0x11a1270>`.

When we call the function `drawSquare`, we pass it two objects, `t` and `150`. When a function is called, the objects are matched up with the parameter names they were given when we defined the function. The first parameter in the list names the first object, the second parameter in the list names the second object, and so on. This means that inside the `drawSquare` function `t` goes by the name `myTurtle`. The turtle object now has two names. Similarly, inside the `drawSquare` function, 150 now has the name `sideLength`.

Figure 1.11 shows a reference diagram to help you understand how all the names and objects are matched up. This diagram is simplified somewhat, but we will add more details to such diagrams in later chapters. The important thing to notice is the relationship between the names of the parameters of a function and the objects that are passed to the function when it is called.

In particular, notice the difference in location between the `Turtle` constructor and the `drawSquare` function. When we execute the statement `import turtle`, we are creating a module, and any functions inside the module are "hidden" from us unless we use the `turtle` prefix. However, if we use the statement `from ds import *`, all the functions defined in the file `ds.py` are visible to us without the `ds` prefix.

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION

In Figure 1.11 there are three names: (1) `turtle`, which references the turtle module we imported previously; (2) `t`, which references the turtle object we created with the call to the turtle constructor (`turtle.Turtle()`). You can see the turtle constructor inside the `turtle` module. (3) `drawSquare` references the function object we defined for drawing squares. In the function object named `drawSquare`, you can see that there are two additional names: (1) `myTurtle`, which references the turtle object named `t`, and (2) `sideLength`, which references the integer object 150 and has no other name.

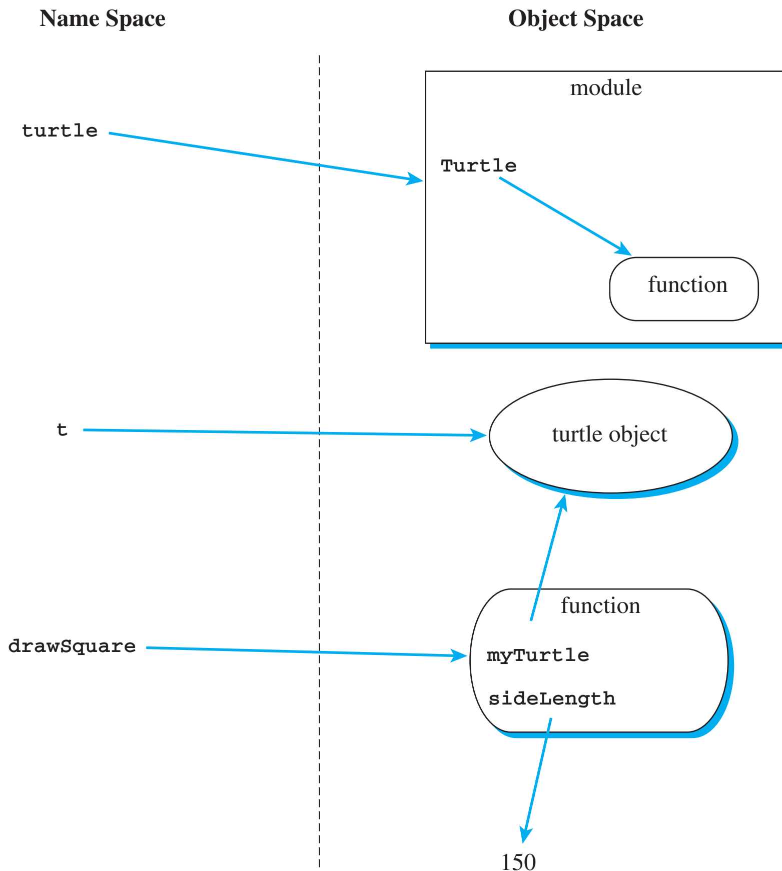

Figure 1.11 A reference diagram for the function `drawSquare`

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION

Exercises

1.25 Modify the `drawSquare` function to draw a rectangle whose width is twice the `sideLength`.

1.26 Create a new function called `drawRectangle` that takes three parameters: `myTurtle`, `width`, and `height`.

1.27 Suppose that in Session 1.8 we had used the expression `import ds` instead of `from ds import *`. Continue the rest of the session. Draw a reference diagram that illustrates this session.

1.28 Call your function as follows: `drawRectangle(t,50,300)`.

1.29 Draw a reference diagram for the previous problem.

1.5.4 Repetition

Although the `drawSquare` function we discussed earlier worked just fine, there is one thing that is unsatisfying about our solution. We had to duplicate the move forward and turn to the right statements four times. In order to eliminate this duplication, Python provides us with a statement that allows us to repeat a block of code multiple times. This statement is called a *for* loop. The *for* loop is an example of using *repetition* in our program.

Before we rewrite our `drawSquare` function, let's look at the structure of a *for* loop to get a better idea of how this statement works:

```
for i in range(n):
    statement1
    statement2
    ...
```

Notice that the *for* loop template has some similarity to the *def* template. There is a beginning line ending with a colon, followed by an indented block of code. All you need to know about a *for* loop at this point is that each statement in the indented block will be evaluated `n` times, where `n` is the parameter to the `range` function. We will talk more about the `range` function as well as the `i` and *in* part of the statement shortly. If the first line was `for i in range(10):`, then each statement would be evaluated ten times.

Now let's see how we can use this idea of simple repetition to make our `drawSquare` function more elegant. In fact, all we are going to do is wrap the two lines that tell the turtle to

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION

move forward and turn right inside a loop that will be evaluated four times. You can see the new and improved `drawSquare` function in Listing 1.3.

```
1 def drawSquare(myTurtle,sideLength):
2     for i in range(4):
3         myTurtle.forward(sideLength)
4         myTurtle.right(90)
```

Listing 1.3 A better version of the `drawSquare` function

Drawing a Spiral

Our goal for this section is to understand how we can use the *for* loop to have the turtle create a more complicated square spiral pattern. To create a spiral pattern, we need to have the sides of our square grow each time the turtle moves forward. We can make that happen easily when we understand the two components from the *for* statement we ignored a moment ago. First, let's look at the `range` function. If you ask Python to evaluate the expression `range(5)`, you will get back an object representing the sequence of numbers `[0, 1, 2, 3, 4]`.

The `range` function is very versatile in that it can create all kinds of sequences depending on the parameters that we supply. There are three ways we can call `range`:

- **range(stop):** Creates a sequence of numbers starting at `0` going up to `stop-1`.
- **range(start, stop):** Creates a sequence of numbers beginning at `start` going up to `stop-1`.
- **range(start, stop, step):** Creates a sequence of numbers beginning at `start` going up to `stop-1` counting by `step`.

Exercises

1.30 Use the `range` function to create a sequence of the multiples of 5 up to 50.

1.31 Use the `range` function to create a sequence of numbers from −10 to 10.

1.32 Use the `range` function to create a sequence of numbers from 10 to −10.

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION

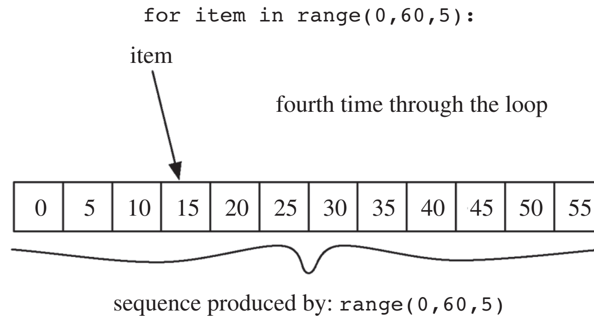

Figure 1.12 A loop variable naming each item in a sequence

Now that you have a better understanding of the `range` function, let's look at the **loop variable**—the variable that always follows the keyword *for*. In Listing 1.3 the loop variable is named `i`. The way the *for* loop works in Python is that the loop variable starts out as the name for the first item in the sequence produced by the `range` function the first time through the loop. The second time through, the loop variable is the name for the second item in the sequence and so on until there are no more items in the sequence. We can use any valid Python name for the loop variable.

Figure 1.12 shows the sequence produced by the call to `range` and the item in the list named by `item` during the fourth repetition of the loop. Note that 55 is the last item in the sequence because the upper bound, 60, is never included in the result of the `range` function.

Loop variables can be used in expressions and function calls like any other Python names. To solve the spiral problem, we will use a loop variable in a function that draws a spiral. The parameters to this function will be the turtle and a bound for the longest side on the spiral. We will start with the first side having length 1, then each time through the loop we will increase the length of the side of the spiral by 5. Listing 1.4 shows the function `drawSpiral`.

```
1  def drawSpiral(myTurtle,maxSide):
2      for sideLength in range(1,maxSide+1,5):
3          myTurtle.forward(sideLength)
4          myTurtle.right(90)
```

Listing 1.4 A Python function to draw a spiral

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION

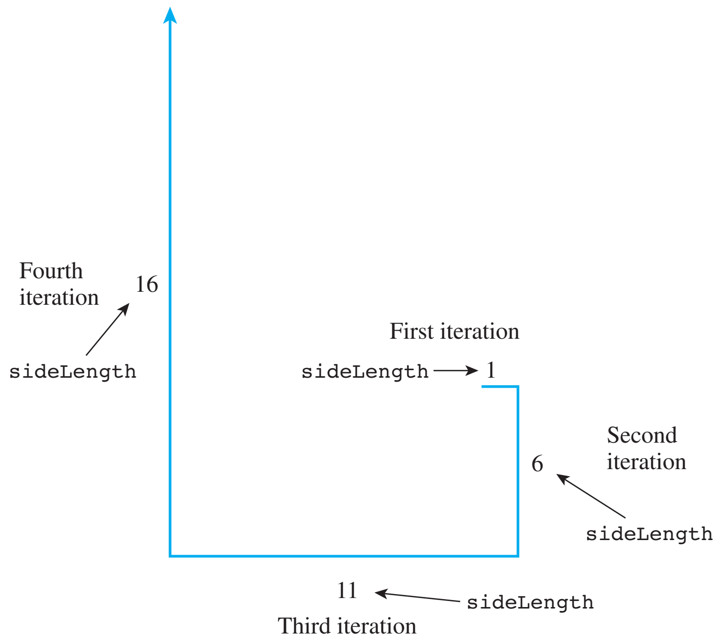

Figure 1.13 The first four iterations of the loop in `drawSpiral(t,150)`

Figure 1.13 illustrates the first four iterations of the *for* loop in Listing 1.4. In the first iteration, the turtle draws a line that is 1 unit long because `sideLength` refers to the first item in the sequence produced by `range(1,maxSide+1,5)`, which is 1. In the second iteration, `sideLength` refers to the number 6, so the turtle draws a line that is 6 units long. In the third iteration, `sideLength` refers to 11; in the fourth iteration, `sideLength` refers to the fourth number in the sequence, which is 16.

Exercises

1.33 Modify the spiral function to turn more than 90 degrees for each iteration.

1.34 Modify the spiral function to turn less than 90 degrees for each iteration.

1.35 Modify the spiral function to use the loop variable as the number of degrees to turn.

1.36 Modify the spiral function to use a second turtle and create two spirals in opposite directions.

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION

1.37 Write a function `drawTriangle` that takes two side lengths and the angle between them and draws the triangle. (Hint: You need to remember the starting point.)

1.38 Write a function that draws a series of 10 squares, with each square being 5 pixels smaller on each side. The squares should all start at the same location.

1.39 Redo the last question so that the squares are all centered.

1.40 Use the turtle to plot the function $y = x^2$.

1.41 Use the turtle to plot the function $y = \frac{x}{2} + 3$.

Drawing a Circle

Our final problem for this chapter is to write a function that uses a turtle to draw a circle of a given radius. Although this may seem like a daunting task, we will help ourselves by solving a simpler problem first and then using what we learn to solve the more general problem. The first thing we must recognize is that the turtle's functionality allows us to draw only straight lines. We will approximate a curved line by drawing many short straight lines.

Suppose that our problem was to draw a triangle rather than a circle. Starting from the `drawSquare` function, it is not too hard to imagine how we would modify `drawSquare` to write `drawTriangle`. We would change the call to `range(4)` to be `range(3)` since we only need three sides. The other change we need to make is the number of degrees we pass as the parameter for our right turn function.

How many degrees do we need to turn each time to draw an equilateral triangle? When we are all done, we want our turtle to be pointed in the same direction it was when we started. That means that as we draw the three sides of our triangle the turtle is going to turn through 360 degrees. That matches our `drawSquare` function where we made four 90 degree turns ($4 \cdot 90 = 360$). So to draw a triangle we will use $360 \div 3 = 120$ for our turning angle. Listing 1.5 shows the small changes made to the `drawSquare` function.

```
1  def drawTriangle(myTurtle,sideLength):
2      for i in range(3):
3          myTurtle.forward(sideLength)
4          myTurtle.right(120)
```

Listing 1.5 A Python function to draw a triangle

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION

Name	Sides	range()	Angle
Triangle	3	3	$360/3 = 120$
Square	4	4	$360/4 = 90$
Pentagon	5	5	$360/5 = 72$
Octagon	8	8	$360/8 = 45$

Table 1.4 Number of sides versus turning angle for several polygons

We now have two simple examples of polygons, the equilateral triangle and the square. It is relatively easy to imagine how to write a function to draw a pentagon or an octagon by following the pattern we have established with the triangle and square. In fact, Table 1.4 illustrates the values we would supply to the `range` and `right` functions for several different polygons.

What Table 1.4 suggests is that we can write a function that is more abstract than `drawSquare`, `drawTriangle`, or even `drawOctagon`. The abstraction is a regular polygon. Creating one function that can replace many simpler functions at a higher level of abstraction is a common and important problem-solving technique in computer science.

Given what we have learned, we can write a function that draws any regular polygon if we pass the function a third parameter. The third parameter tells the function how many sides we want. Once we know how many sides are required, we can easily have Python calculate the turning angle for us using the formula `turnAngle = 360 / numSides`. You can see the new `drawPolygon` function in Listing 1.6.

```
1  def drawPolygon(myTurtle,sideLength,numSides):
2      turnAngle = 360 / numSides
3      for i in range(numSides):
4          myTurtle.forward(sideLength)
5          myTurtle.right(turnAngle)
```

Listing 1.6 A Python function to draw any size regular polygon

Session 1.9 along with Figure 1.14 illustrate several calls to the `drawPolygon` function. Note that the call `drawPolygon(t,20,20)` makes a pretty good approximation of a circle. Note also that we are assuming that the `drawPolygon` function has been saved in a file called `dp.py`.

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION

```
>>> from dp import *
>>> import turtle
>>> t = turtle.Turtle()
>>> t.up()
>>> t.backward(200)
>>> t.left(90)
>>> t.down()
>>> drawPolygon(t,100,4)
>>> drawPolygon(t,100,8)
>>> drawPolygon(t,50,20)
>>> drawPolygon(t,20,20)
```

Session 1.9 Testing the `drawPolygon` function

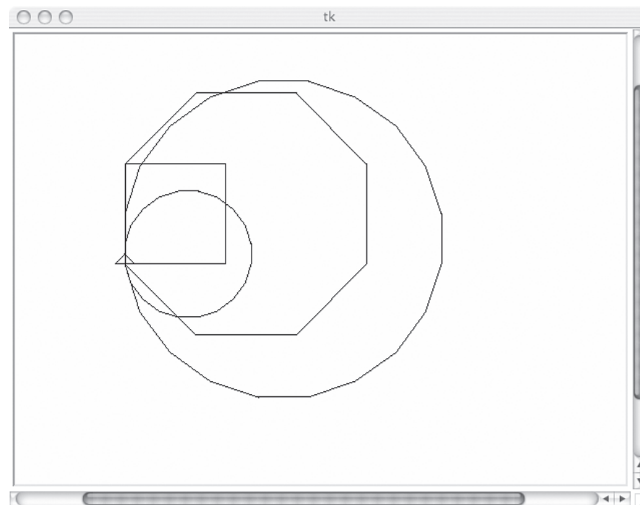

Figure 1.14 Several polygons drawn by the `drawPolygon` function

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION

We have worked our way through the problem of drawing a circle by using a large degree polygon as an approximation. Now let's return to the original problem statement, which asks us to draw a circle of a given radius. So, the last trick is to figure out how to use the radius to compute the number of sides and the side length.

Suppose that to get the smoothest circle possible, even a very large circle, we choose to always have 360 sides to our polygon. This means that the turtle will always turn 1 degree and should give us a smooth circle even for a large radius.

Only one question remains: How do we decide what side length to use for a given radius? One good approximation is to use the relationship of the radius of a circle to the circumference. If we are doing a good job of approximating a circle, then the circumference of a circle should be very close to the sum of the individual sides of the polygon. Recall that the circumference of a circle can be calculated from the radius using the formula $circumference = 2 \cdot \pi \cdot radius$. Once we have the circumference, we can calculate an individual side length by dividing by the number of sides, which we have decided will be 360. Listing 1.7 shows the `drawCircle` function, which makes use of our `drawPolygon` function to draw a circle with a given radius.

```
1  def drawCircle(myTurtle,radius):
2      circumference = 2 * 3.1415 * radius
3      sideLength = circumference / 360
4      drawPolygon(myTurtle,sideLength,360)
```

Listing 1.7 A Python function to draw a circle

The `drawCircle` function is extremely simple because we have reduced the problem of drawing a circle to drawing a polygon with a particular number of sides and side lengths. We first calculate the circumference and then calculate the side length given the circumference and number of sides. Since we already know how to draw a polygon, we do not have to redo that work. We can build on what we have already done and use the `drawPolygon` function. Session 1.10 and Figure 1.15 illustrate the use of the `drawCircle` function to draw both a large and a small circle.

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION

```
>>> from dp import *
>>> import turtle
>>> t = turtle.Turtle()
>>> t.up()
>>> t.backward(200)
>>> t.left(90)
>>> t.down()
>>> drawCircle(t,20)
>>> drawCircle(t,200)
>>>
```

Session 1.10 Drawing a circle

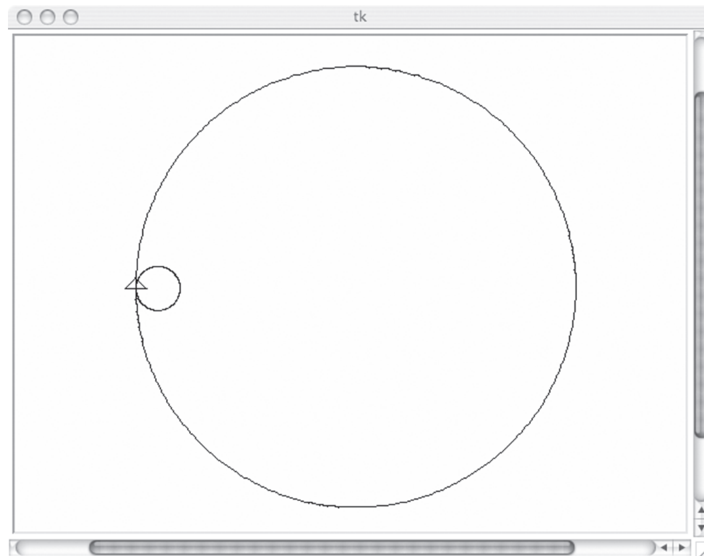

Figure 1.15 Drawing a circle for a given radius

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION

Exercises

1.42 Modify the `drawCircle` function so that the circle is drawn with the center at the turtle's present position.

1.43 The `drawCircle` function is somewhat inefficient: for small circles, 360 sides is really overkill and for very large circles 360 sides might be too few. See if you can devise a way to make the number of sides and turning angle dependent on the radius so that smaller circles use fewer sides and larger circles use more.

1.6 Summary

This chapter introduced the following fundamental building blocks of programming and Python:

- Primitive types
- Expressions
- Naming objects
- Using modules and functions provided by Python
- Writing your own functions to extend the functionality provided by Python

In addition, the approach we followed to use the turtle to draw a circle illustrates an important problem-solving pattern that you will use many times as you progress through this book. That pattern can be summarized as follows:

- Simplify in order to understand the problem better.
- Generalize to solve many problems with one function.
- Build on what you have learned to solve more complex problems.

In subsequent chapters we will continue to use these basic building blocks. There will be more tools to add to your toolbox. We have glossed over a few details on some of the ideas introduced in this chapter but will return to them later. Keep in mind the idea behind this book: to focus on problem solving while continually adding to your knowledge of programming and computer science.

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION

Key Terms

abstraction
algorithm
assignment statement
block
complex number
computer science
constructor
dereferencing
dot operator
expression
floating-point number
generalization
instance
integer
keywords
list
loop variable
module
name
object
parameter
program
programming language
prompt
Python
repetition
representation
sequence object
simplification
type
variable

Python Keywords

`def` `for` `from` `import` `in` `range`

Bibliography

[BES05] Chris L. Barrett, Stephen G. Eubank, and James P. Smith. If smallpox strikes portland. *Scientific American*, March 2005.

[Eco06] Economist. The physical internet. *The Economist*, June 2006.

[fold] Stanford folding@home project. http://folding.stanford.edu

[How97] Kenneth R. Howard. Unjamming traffic with computers. *Scientific American*, 1997.

[Zei99] Paul Zeitz. *The Art and Craft of Problem Solving*. Wiley, 1999.

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION

Programming Exercises

1.1 Using the `drawSquare` function, you can have the turtle draw an interesting flowerlike shape by drawing many squares. Each square is drawn after turning the turtle by some number of degrees between each square. Write a function `drawflower` that takes the number of squares to draw as a parameter (`numSquares`) and draws a flower by repeating the square `numSquares` times. You will need to figure out how far to turn the turtle based on `numSquares`.

1.2 Write a function to make the turtle draw a five-pointed star.

1.3 Write a function to make the turtle draw an n-pointed star when n is restricted to be odd.

1.4 Write a function to have the turtle draw a simple line drawing of anything you want.

1.5 There are two more functions that the turtle understands: `begin_fill()` and `end_fill()`. When `begin_fill` is called, the turtle keeps track of its starting point, and all the lines it has drawn until `end_fill` is called. When `end_fill` is called, the turtle fills in the space enclosed by the lines that the turtle has drawn. Use these new functions to draw a more interesting picture.

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION