THIRD EDITION

PYTHON PROGRAMMING:

AN INTRODUCTION TO COMPUTER SCIENCE

JOHN ZELLE

FRANKLIN, BEEDLE
[INDEPENDENT PUBLISHERS SINCE 1985]

Chapter 1 Computers and Programs

Objectives

- To understand the respective roles of hardware and software in computing systems.
- To learn what computer scientists study and the techniques that they use.
- To understand the basic design of a modern computer.
- To understand the form and function of computer programming languages.
- To begin using the Python programming language.
- To learn about chaotic models and their implications for computing.

1.1 The Universal Machine

Almost everyone has used a computer at one time or another. Perhaps you have played computer games or used a computer to write a paper, shop online, listen to music, or connect with friends via social media. Computers are used to predict the weather, design airplanes, make movies, run businesses, perform financial transactions, and control factories.

Have you ever stopped to wonder what exactly a computer is? How can one device perform so many different tasks? These basic questions are the starting point for learning about computers and computer programming.

A modern computer can be defined as "a machine that stores and manipulates information under the control of a changeable program." There are two key elements to this definition. The first is that computers are devices for manipulating information. This means we can put information into a computer, and it can transform the information into new, useful forms, and then output or display the information for our interpretation.

Computers are not the only machines that manipulate information. When you use a simple calculator to add up a column of numbers, you are entering information (the numbers) and the calculator is processing the information to compute a running sum which is then displayed. Another simple example is a gas pump. As you fill your tank, the pump uses certain inputs: the current price of gas per gallon and signals from a sensor that reads the rate of gas flowing into your car. The pump transforms this input into information about how much gas you took and how much money you owe.

We would not consider either the calculator or the gas pump as full-fledged computers, although modern versions of these devices may actually contain embedded computers. They are different from computers in that they are built to perform a single, specific task. This is where the second part of our definition comes into the picture: Computers operate under the control of a changeable program. What exactly does this mean?

A *computer program* is a detailed, step-by-step set of instructions telling a computer exactly what to do. If we change the program, then the computer performs a different sequence of actions, and hence, performs a different task. It is this flexibility that allows your PC to be at one moment a word processor, at the next moment a financial planner, and later on, an arcade game. The machine stays the same, but the program controlling the machine changes.

Every computer is just a machine for *executing* (carrying out) programs. There are many different kinds of computers. You might be familiar with Macintoshes, PCs, laptops, tablets and smartphones, but there are literally thousands of other kinds of computers both real and theoretical. One of the remarkable discoveries of computer science is the realization that all of these different computers have the same power; with suitable programming, each computer can basically do all the things that any other computer can do. In this sense, the PC that you might have sitting on your desk is really a universal machine. It can do anything you want it to do, provided you can describe the task to be accomplished in sufficient detail. Now that's a powerful machine!

1.2 Program Power

You have already learned an important lesson of computing: *Software* (programs) rules the *hardware* (the physical machine). It is the software that determines what any computer can do. Without software, computers would just be expensive paperweights. The process of creating software is called *programming*, and that is the main focus of this book.

Computer programming is a challenging activity. Good programming requires an ability to see the big picture while paying attention to minute detail. Not everyone has the talent to become a first-class programmer, just as not everyone has the skills to be a professional athlete. However, virtually anyone *can* learn how to program computers. With some patience and effort on your part, this book will help you to become a programmer.

There are lots of good reasons to learn programming. Programming is a fundamental part of computer science and is, therefore, important to anyone interested in becoming a computer professional. But others can also benefit from the experience. Computers have become a commonplace tool in our society. Understanding the strengths and limitations of this tool requires an understanding of programming. Non-programmers often feel they are slaves of their computers. Programmers, however, are truly in control. If you want to become a more intelligent user of computers, then this book is for you.

Programming can also be loads of fun. It is an intellectually engaging activity that allows people to express themselves through useful and sometimes remarkably beautiful creations. Believe it or not, many people actually write computer programs as a hobby. Programming also develops valuable problem-solving skills, especially the ability to analyze complex systems by reducing them to interactions of understandable subsystems.

As you probably know, programmers are in great demand. More than a few liberal arts majors have turned a couple of computer programming classes into a lucrative career option. Computers are so commonplace in the business world today that the ability to understand and program computers might just give you the edge over your competition regardless of your occupation. When inspiration strikes, you could be poised to write the next killer app.

1.3 What Is Computer Science?

You might be surprised to learn that computer science is not the study of computers. A famous computer scientist named Edsger Dijkstra once quipped that computers are to computer science what telescopes are to astronomy. The computer is an important tool in computer science, but it is not itself the object of study. Since a computer can carry out any process that we can describe, the real question is "What processes can we describe?" To put it another way, the fundamental question of computer science is simply "What can be computed?" Computer scientists use numerous techniques of investigation to answer this question. The three main ones are *design, analysis,* and *experimentation*.

One way to demonstrate that a particular problem can be solved is to actually design a solution. That is, we develop a step-by-step process for achieving the desired result. Computer scientists call this an *algorithm*. That's a fancy word that basically means "recipe." The design of algorithms is one of the most important facets of computer science. In this book you will find techniques for designing and implementing algorithms.

One weakness of design is that it can only answer the question "What is computable?" in the positive. If I can devise an algorithm, then the problem is solvable. However, failing to find an algorithm does not mean that a problem is unsolvable. It may mean that I'm just not smart enough, or I haven't hit upon the right idea yet. This is where analysis comes in.

Analysis is the process of examining algorithms and problems mathematically. Computer scientists have shown that some seemingly simple problems are not solvable by *any* algorithm. Other problems are *intractable*. The algorithms that solve these problems take too long or require too much memory to be of practical value. Analysis of algorithms is an important part of computer science; throughout this book we will touch on some of the fundamental principles. Chapter 13 has examples of unsolvable and intractable problems.

Some problems are too complex or ill-defined to lend themselves to analysis. In such cases, computer scientists rely on experimentation; they actually implement systems and then study the resulting behavior. Even when theoretical analysis is done, experimentation is often needed in order to verify and refine the analysis. For most problems, the bottom line is whether a working, reliable system can be built. Often we require empirical testing of the system to determine that this bottom line has been met. As you begin writing your own programs, you will get plenty of opportunities to observe your solutions in action.

I have defined computer science in terms of designing, analyzing, and evaluating algorithms, and this is certainly the core of the academic discipline. These days, however, computer scientists are involved in far-flung activities, all of which fall under the general umbrella of computing. Some examples

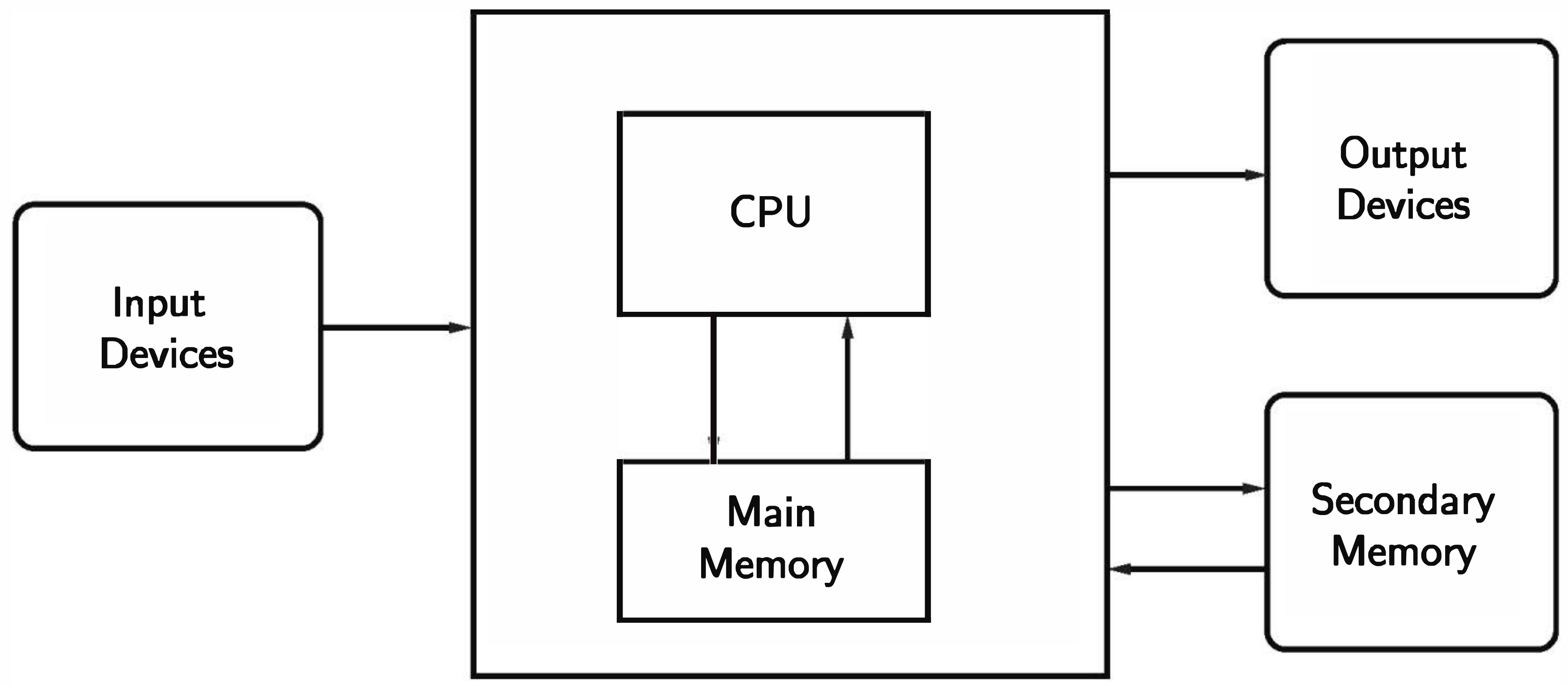


Figure 1.1: Functional view of a computer

include mobile computing, networking, human-computer interaction, artificial intelligence, computational science (using powerful computers to model scientific processes), databases and data mining, software engineering, web and multimedia design, music production, management information systems, and computer security. Wherever computing is done, the skills and knowledge of computer science are being applied.

1.4 Hardware Basics

You don't have to know all the details of how a computer works to be a successful programmer, but understanding the underlying principles will help you master the steps we go through to put our programs into action. It's a bit like driving a car. Knowing a little about internal combustion engines helps to explain why you have to do things like fill the gas tank, start the engine, step on the accelerator, and so on. You could learn to drive by just memorizing what to do, but a little more knowledge makes the whole process much more understandable. Let's take a moment to "look under the hood" of your computer.

Although different computers can vary significantly in specific details, at a higher level all modern digital computers are remarkably similar. Figure 1.1 shows a functional view of a computer. The *central processing unit* (CPU) is the "brain" of the machine. This is where all the basic operations of the computer are carried out. The CPU can perform simple arithmetic operations like adding two numbers and can also do logical operations like testing to see if two numbers are equal.

The memory stores programs and data. The CPU can directly access only information that is stored in *main memory* (called RAM for *Random Access Memory*). Main memory is fast, but it is also volatile. That is, when the power is turned off, the information in the memory is lost. Thus, there must also be some secondary memory that provides more permanent storage.

In a modern personal computer, the principal secondary memory is typically an internal hard disk drive (HDD) or a solid state drive (SSD). An HDD stores information as magnetic patterns on a spinning disk, while an SSD employs electronic circuits known as flash memory. Most computers also support removeable media for secondary memory such as USB memory "sticks" (also a form of flash memory) and DVDs (digital versatile discs), which store information as optical patterns that are read and written by a laser.

Humans interact with the computer through input and output devices. You are probably familiar with common devices such as a keyboard, mouse, and monitor (video screen). Information from input devices is processed by the CPU and may be shuffled off to the main or secondary memory. Similarly, when information needs to be displayed, the CPU sends it to one or more output devices.

So what happens when you fire up your favorite game or word processing program? First, the instructions that comprise the program are copied from the (more) permanent secondary memory into the main memory of the computer. Once the instructions are loaded, the CPU starts executing the program.

Technically the CPU follows a process called the *fetch-execute cycle*. The first instruction is retrieved from memory, decoded to figure out what it represents, and the appropriate action carried out. Then the next instruction is fetched, decoded, and executed. The cycle continues, instruction after instruction. This is really all the computer does from the time that you turn it on until you turn it off again: fetch, decode, execute. It doesn't seem very exciting, does it? But the computer can execute this stream of simple instructions with blazing speed, zipping through billions of instructions each second. Put enough simple instructions together in just the right way, and the computer does amazing things.

1.5 Programming Languages

Remember that a program is just a sequence of instructions telling a computer what to do. Obviously, we need to provide those instructions in a language that a computer can understand. It would be nice if we could just tell a computer what to do using our native language, like they do in science fiction movies. ("Computer, how long will it take to reach planet Alphalpha at maximum warp?") Computer scientists have made great strides in this direction; you may be familiar with technologies such as Siri (Apple), Google Now (Android), and Cortana (Microsoft). But as anyone who has seriously useded such systems can attest, designing a computer program to fully understand human language is still an unsolved problem.

Even if computers could understand us, human languages are not very well suited for describing complex algorithms. Natural language is fraught with ambiguity and imprecision. For example, if I say "I saw the man in the park with the telescope," did I have the telescope, or did the man? And who was in the park? We understand each other most of the time only because all humans share a vast store of common knowledge and experience. Even then, miscommunication is commonplace.

Computer scientists have gotten around this problem by designing notations for expressing computations in an exact and unambiguous way. These special notations are called *programming languages*. Every structure in a programming language has a precise form (its *syntax*) and a precise meaning (its *semantics*). A programming language is something like a code for writing down the instructions that a computer will follow. In fact, programmers often refer to their programs as *computer code*, and the process of writing an algorithm in a programming language is called *coding*.

Python is one example of a programming language and is the language that we will use throughout this book.[1] You may have heard of some other commonly used languages, such as C++, Java, Javascript, Ruby, Perl, Scheme, or BASIC. Computer scientists have developed literally thousands of programming languages, and the languages themselves evolve over time yielding multiple, sometimes very different, versions. Although these languages differ in many details, they all share the property of having well-defined, unambiguous syntax and semantics.

All of the languages mentioned above are examples of *high-level* computer languages. Although they are precise, they are designed to be used and understood by humans. Strictly speaking, computer hardware can understand only a very low-level language known as *machine language*.

Suppose we want the computer to add two numbers. The instructions that the CPU actually carries out might be something like this:

[1]This edition of the text was developed and tested using Python version 3.4. Python 3.5 is now available. If you have an earlier version of Python installed on your computer, you should upgrade to the latest stable 3.x version to try out the examples.

```
load the number from memory location 2001 into the CPU
load the number from memory location 2002 into the CPU
add the two numbers in the CPU
store the result into location 2003
```

This seems like a lot of work to add two numbers, doesn't it? Actually, it's even more complicated than this because the instructions and numbers are represented in *binary* notation (as sequences of 0s and 1s).

In a high-level language like Python, the addition of two numbers can be expressed more naturally: `c = a + b`. That's a lot easier for us to understand, but we need some way to translate the high-level language into the machine language that the computer can execute. There are two ways to do this: a high-level language can either be *compiled* or *interpreted*.

A *compiler* is a complex computer program that takes another program written in a high-level language and translates it into an equivalent program in the machine language of some computer. Figure 1.2 shows a block diagram of the compiling process. The high-level program is called *source code*, and the resulting *machine code* is a program that the computer can directly execute. The dashed line in the diagram represents the execution of the machine code (also known as "running the program").

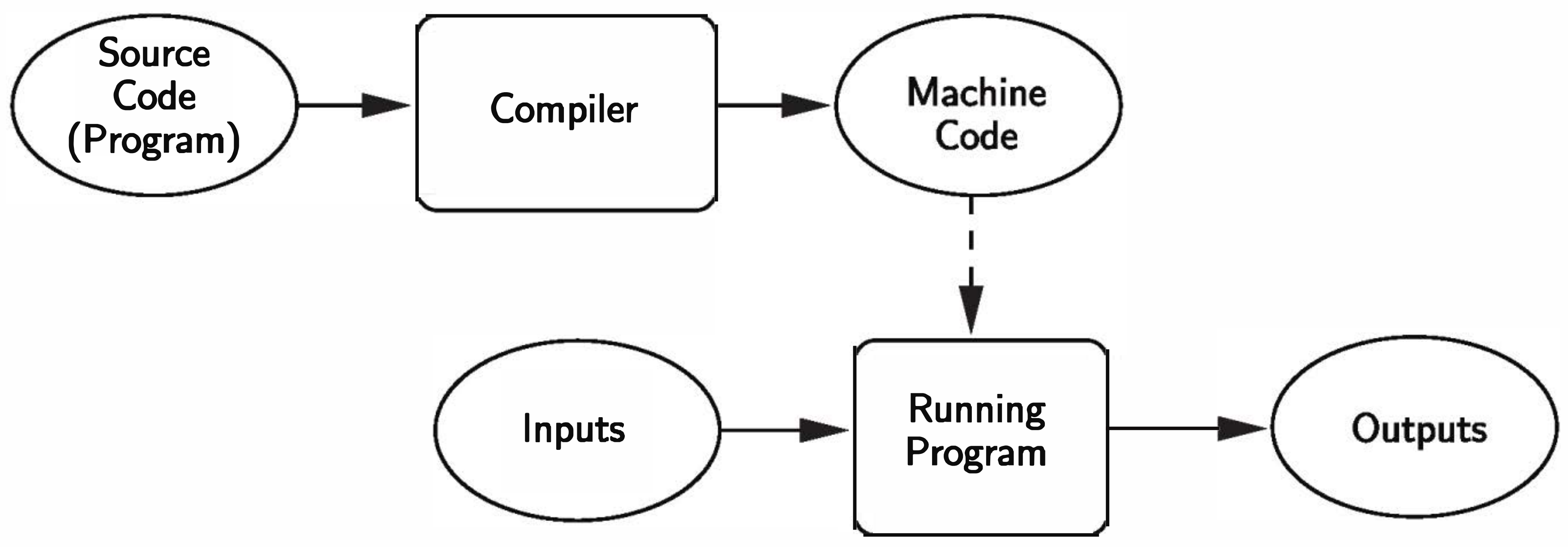


Figure 1.2: Compiling a high-level language

An *interpreter* is a program that simulates a computer that understands a high-level language. Rather than translating the source program into a machine language equivalent, the interpreter analyzes and executes the source code instruction by instruction as necessary. Figure 1.3 illustrates the process.

The difference between interpreting and compiling is that compiling is a one-shot translation; once a program is compiled, it may be run over and over again without further need for the compiler or the source code. In the interpreted

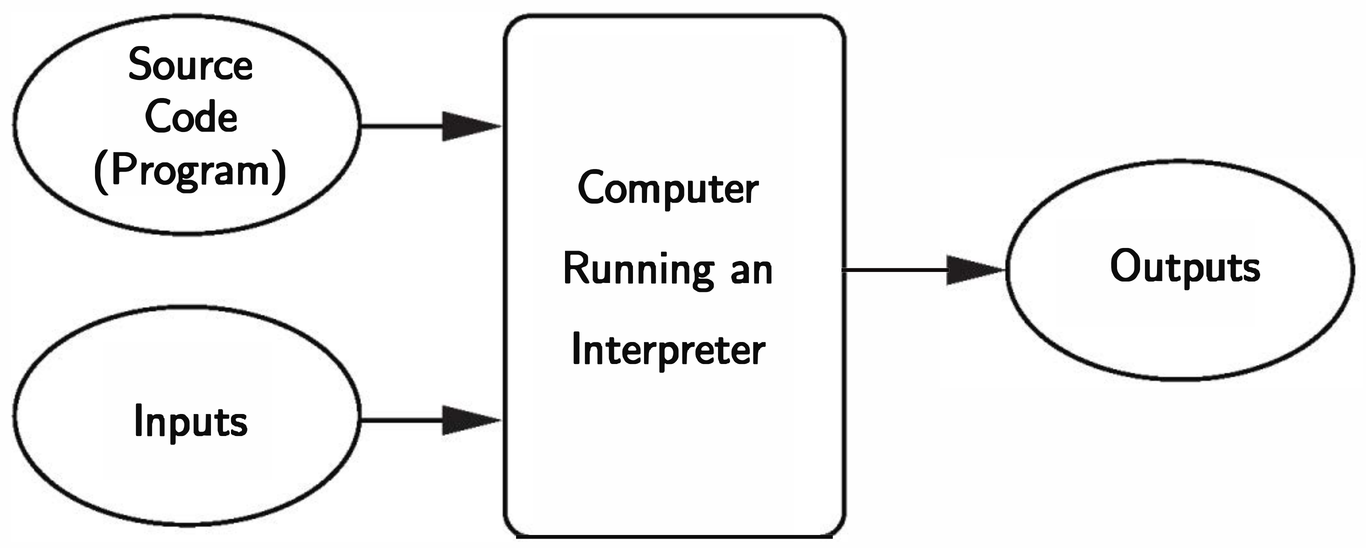


Figure 1.3: Interpreting a high-level language

case, the interpreter and the source are needed every time the program runs. Compiled programs tend to be faster, since the translation is done once and for all, but interpreted languages lend themselves to a more flexible programming environment as programs can be developed and run interactively.

The translation process highlights another advantage that high-level languages have over machine language: *portability*. The machine language of a computer is created by the designers of the particular CPU. Each kind of computer has its own machine language. A program for an Intel i7 Processor in your laptop won't run directly on an ARMv8 CPU in your smartphone. On the other hand, a program written in a high-level language can be run on many different kinds of computers as long as there is a suitable compiler or interpreter (which is just another program). As a result, I can run the exact same Python program on my laptop and my tablet; even though they have different CPUs, they both sport a Python interpreter.

1.6 The Magic of Python

Now that you have all the technical details, it's time to start having fun with Python. The ultimate goal is to make the computer do our bidding. To this end, we will write programs that control the computational processes inside the machine. You have already seen that there is no magic in this process, but in some ways programming *feels* like magic.

The computational processes inside the computer are like magical spirits that we can harness for our work. Unfortunately, those spirits only understand a very arcane language that we do not know. What we need is a friendly genie that can direct the spirits to fulfill our wishes. Our genie is a Python interpreter. We can give instructions to the Python interpreter, and it directs the underlying spirits to carry out our demands. We communicate with the genie through a special language of spells and incantations (i.e., Python). The best way to start learning about Python is to let our genie out of the bottle and try some spells.

With most Python installations, you can start a Python interpreter in an interactive mode called a *shell*. A shell allows you to type Python commands and then displays the result of executing them. The specifics for starting a shell differ for various installations. If you are using the standard Python distribution for PC or Mac from www.python.org, you should have an application called IDLE that provides a Python shell and, as we'll see later on, also helps you create and edit your own Python programs. The supporting website for this book has information on installing and using Python on a variety of platforms.

When you first launch IDLE (or another Python shell), you should see something like the this:

```
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 24 2015, 22:43:06)
[MSC v.1600 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
```

The exact opening message depends on the version of Python that you are running and the system that you are working on. The important part is the last line; the >>> is a Python *prompt* indicating that our genie (the Python interpreter) is waiting for us to give it a command. In programming languages, a complete command is called a *statement*.

Here is a sample interaction with a Python shell:

```
>>> print("Hello, World!")
Hello, World!
>>> print(2 + 3)
5
>>> print("2 + 3 =", 2 + 3)
2 + 3 = 5
```

Here I have tried out three examples using the Python `print` statement. The first statement asks Python to display the literal phrase `Hello, World!`. Python responds on the next line by printing the phrase. The second `print` statement asks Python to print the sum of 2 and 3. The third `print` combines these two ideas. Python prints the part in quotes, `2 + 3 =`, followed by the result of adding 2 + 3, which is 5.

This kind of shell interaction is a great way to try out new things in Python. Snippets of interactive sessions are sprinkled throughout this book. When you see the Python prompt >>> in an example, that should tip you off that an interactive session is being illustrated. It's a good idea to fire up your own Python shell and try the examples.

Usually we want to move beyond one-line snippets and execute an entire sequence of statements. Python lets us put a sequence of statements together to create a brand-new command or *function*. Here is an example of creating a new function called `hello`:

```
>>> def hello():
        print("Hello")
        print("Computers are fun!")

>>>
```

The first line tells Python that we are *defining* a new function and we are naming it `hello`. The following lines are indented to show that they are part of the `hello` function. (*Note*: Some shells will print ellipses ["..."] at the beginning of the indented lines). The blank line at the end (obtained by hitting the `<Enter>` key twice) lets Python know that the definition is finished, and the shell responds with another prompt. Notice that typing the definition did not cause Python to print anything yet. We have told Python what *should* happen when the `hello` function is used as a command; we haven't actually asked Python to perform it yet.

A function is *invoked* (or *called*) by typing its name followed by parentheses. Here's what happens when we use our `hello` command:

```
>>> hello()
Hello
Computers are fun!
>>>
```

Do you see what this does? The two `print` statements from the `hello` function definition are executed in sequence.

You may be wondering about the parentheses in the definition and use of `hello`. Commands can have changeable parts called *parameters* (also called *arguments*) that are placed within the parentheses. Let's look at an example of a customized greeting using a parameter. First the definition:

```
>>> def greet(person):
        print("Hello", person)
        print("How are you?")
```

Now we can use our customized greeting.

```
>>> greet("John")
Hello John
How are you?
>>> greet("Emily")
Hello Emily
How are you?
>>>
```

Can you see what is happening here? When using `greet` we can send different names to customize the result. You might also notice that this looks similar to the `print` statements from before. In Python, `print` is an example of a built-in function. When we call the `print` function, the parameters in the parentheses tell the function what to print.

We will discuss parameters in detail later on. For the time being the important thing to remember is that the parentheses must be included after the function name whenever we want to execute a function. This is true even when no parameters are given. For example, you can create a blank line of output using `print` without any parameters.

```
>>> print()

>>>
```

But if you type just the name of the function, omitting the parentheses, the function will not actually execute. Instead, an interactive Python session will show some output indicating what function that name refers to, as this interaction shows:

```
>>> greet
<function greet at 0x8393aec>
>>> print
<built-in function print>
```

The funny text `0x8393aec` is the location (address) in computer memory where the `greet` function definition happens to be stored. If you are trying this out on your own computer, you will almost certainly see a different address.

One problem with entering functions interactively into a Python shell as we did with the `hello` and `greet` examples is that the definitions are lost when we quit the shell. If we want to use them again the next time, we have to type them all over again. Programs are usually created by typing definitions into a separate file called a *module* or *script*. This file is saved in secondary memory so that it can be used over and over again.

A module file is just a file of text, and you can create one using any application for editing text, such as notepad or a word processor, provided you save your program as a "plain text" file. A special type of application known as an *Integrated Development Environment* (IDE) simplifies the process. An IDE is specifically designed to help programmers write programs and includes features such as automatic indenting, color highlighting, and interactive development. IDLE is a good example. So far we have just been using IDLE as a Python shell, but it is actually a simple but complete development environment.[2]

Let's illustrate the use of a module file by writing and running a complete program. Our program will explore a mathematical concept known as chaos. To type this program into IDLE, you should select the *File/New File* menu option. This brings up a blank (non-shell) window where you can type a program. Here is the Python code for our program:

```
# File: chaos.py
# A simple program illustrating chaotic behavior.

def main():
    print("This program illustrates a chaotic function")
    x = eval(input("Enter a number between 0 and 1: "))
    for i in range(10):
        x = 3.9 * x * (1 - x)
        print(x)

main()
```

Once you have typed it in, select *File/Save* from the menu and save it with the name `chaos.py`. The `.py` extension indicates that this is a Python module. Be careful where you save your program. Sometimes IDLE starts you out in the system-wide Python folder by default. Make sure to navigate to a folder where you keep your own files. I'd suggest keeping all of your Python programs together in a dedicated folder in your own personal document area.

At this point, you may be trying to make sense out of what you just typed. You can see that this particular example contains lines to define a new function

[2]In fact, IDLE stands for Integrated DeveLopment Environment. The extra "L" is thrown in as a tribute to Eric Idle, of Monty Python fame.

called `main`. (Programs are often placed in a function called `main`.) The last line of the file is the command to invoke this function. Don't worry if you don't understand what `main` actually does; we will discuss it in the next section. The point here is that once we have a program saved in a module file like this, we can run it any time we want.

Our program can be run in a number of different ways that depend on the actual operating system and programming environment that you are using. If you are using a windowing system, you can probably run a Python program by clicking (or double-clicking) on the module file's icon. In a command line situation, you might type a command like `python chaos.py`. When using IDLE you can run a program simply by selecting *Run/Run Module* from the module window menu. Hitting the `<F5>` key is a handy shortcut for this operation.

When IDLE runs the program, control will shift over to the shell window. Here is how that looks:

```
>>> ====================== RESTART ======================
>>>
This program illustrates a chaotic function
Enter a number between 0 and 1: .25
0.73125
0.76644140625
0.6981350104385375
0.8218958187902304
0.5708940191969317
0.9553987483642099
0.166186721954413
0.5404179120617926
0.9686289302998042
0.11850901017563877
>>>
```

The first line is a notification from IDLE indicating that the shell has restarted. IDLE does this each time you run a program so that the program runs in a pristine environment. Python then runs the module from top to bottom, line by line. It's just as if we had typed them one-by-one at the interactive Python prompt. The `def` in the module causes Python to create the `main` function. The last line of this module causes Python to invoke the `main` function, thus running our program. The running program asks the user to enter a number between 0 and 1 (in this case, I typed ".25") and then prints out a series of 10 numbers.

If you browse through the files on your computer, you may notice that Python sometimes creates another folder called `__pycache__` inside the folder where your module files are stored. This is a place where Python stashes companion files with a `.pyc` extension. In this example, Python might create another file called `chaos.pyc`. This is an intermediate file used by the Python interpreter. Technically, Python uses a hybrid compiling/interpreting process. The Python source in the module file is compiled into more primitive instructions called *byte code*. This byte code (the `.pyc`) is then interpreted. Having a `.pyc` file available makes running a module faster the second time around. However, you may delete the byte code files if you wish to save disk space; Python will automatically recreate them as needed.

Running a module under IDLE loads the program into the shell window. You can run the program again by asking Python to execute the `main` command. Simply type the command at the shell prompt. Continuing with our example, here is how it looks when we rerun the program with .26 as the input:

```
>>> main()
This program illustrates a chaotic function
Enter a number between 0 and 1: .26
0.75036
0.73054749456
0.767706625733
0.6954993339
0.825942040734
0.560670965721
0.960644232282
0.147446875935
0.490254549376
0.974629602149
>>>
```

1.7 Inside a Python Program

The output from the `chaos` program may not look very exciting, but it illustrates a very interesting phenomenon known to physicists and mathematicians. Let's take a look at this program line by line and see what it does. Don't worry about understanding every detail right away; we will be returning to all of these ideas in the next chapter.

The first two lines of the program start with the # character:

```
# File: chaos.py
# A simple program illustrating chaotic behavior.
```

These lines are called *comments*. They are intended for human readers of the program and are ignored by Python. The Python interpreter always skips any text from the pound sign (#) through the end of a line.

The next line of the program begins the definition of a function called `main`:

```
def main():
```

Strictly speaking, it would not be necessary to create a `main` function. Since the lines of a module are executed as they are loaded, we could have written our program without this definition. That is, the module could have looked like this:

```
# File: chaos.py
# A simple program illustrating chaotic behavior.

print("This program illustrates a chaotic function")
x = eval(input("Enter a number between 0 and 1: "))
for i in range(10):
    x = 3.9 * x * (1 - x)
    print(x)
```

This version is a bit shorter, but it is customary to place the instructions that comprise a program inside of a function called `main`. One immediate benefit of this approach was illustrated above; it allows us to run the program by simply invoking `main()`. We don't have to restart the Python shell in order to run it again, which would be necessary in the `main`-less case.

The first line inside of `main` is really the beginning of our program.

```
print("This program illustrates a chaotic function")
```

This line causes Python to print a message introducing the program when it runs.

Take a look at the next line of the program:

```
x = eval(input("Enter a number between 0 and 1: "))
```

Here `x` is an example of a *variable*. A variable is used to give a name to a value so that we can refer to it at other points in the program.

The entire line is a statement to get some input from the user. There's quite a bit going on in this line, and we'll discuss the details in the next chapter; for now, you just need to know what it accomplishes. When Python gets to this statement, it displays the quoted message `Enter a number between 0 and 1:` and then pauses, waiting for the user to type something on the keyboard and press the `<Enter>` key. The value that the user types is then stored as the variable `x`. In the first example shown above, the user entered `.25`, which becomes the value of `x`.

The next statement is an example of a *loop*.

```
 for i in range(10):
```

A loop is a device that tells Python to do the same thing over and over again. This particular loop says to do something 10 times. The lines indented underneath the loop heading are the statements that are done 10 times. These form the *body* of the loop.

```
x = 3.9 * x * (1 - x)
print(x)
```

The effect of the loop is exactly the same as if we had written the body of the loop 10 times:

```
x = 3.9 * x * (1 - x)
print(x)
x = 3.9 * x * (1 - x)
print(x)
x = 3.9 * x * (1 - x)
print(x)
x = 3.9 * x * (1 - x)
print(x)
x = 3.9 * x * (1 - x)
print(x)
x = 3.9 * x * (1 - x)
print(x)
x = 3.9 * x * (1 - x)
print(x)
x = 3.9 * x * (1 - x)
print(x)
x = 3.9 * x * (1 - x)
```

```
print(x)
x = 3.9 * x * (1 - x)
print(x)
```

Obviously, using the loop instead saves the programmer a lot of trouble.

But what exactly do these statements do? The first one performs a calculation.

```
x = 3.9 * x * (1 - x)
```

This is called an *assignment* statement. The part on the right side of the = is a mathematical expression. Python uses the * character to indicate multiplication. Recall that the value of x is 0.25 (from the input above). The computed value is 3.9(0.25)(1 − 0.25) or 0.73125. Once the value on the right-hand side is computed, it is saved as (or *assigned to*) the variable that appears on the left-hand side of the =, in this case x. The new value of x (0.73125) replaces the old value (0.25).

The second line in the loop body is a type of statement we have encountered before, a print statement.

```
print(x)
```

When Python executes this statement, the current value of x is displayed on the screen. So the first number of output is 0.73125.

Remember the loop executes 10 times. After printing the value of x, the two statements of the loop are executed again.

```
x = 3.9 * x * (1 - x)
print(x)
```

Of course, now x has the value 0.73125, so the formula computes a new value of x as 3.9(0.73125)(1 − 0.73125), which is 0.76644140625.

Can you see how the current value of x is used to compute a new value each time around the loop? That's where the numbers in the example run came from. You might try working through the steps of the program yourself for a different input value (say 0.5). Then run the program using Python and see how well you did impersonating a computer.

1.8 Chaos and Computers

I said above that the chaos program illustrates an interesting phenomenon. What could be interesting about a screen full of numbers? If you try out the program for yourself, you'll find that, no matter what number you start with, the results are always similar: the program spits back 10 seemingly random numbers between 0 and 1. As the program runs, the value of `x` seems to jump around, well, chaotically.

The function computed by this program has the general form: $k(x)(1-x)$, where k in this case is `3.9`. This is called a logistic function. It models certain kinds of unstable electronic circuits and is also sometimes used to model population variation under limiting conditions. Repeated application of the logistic function can produce chaos. Although our program has a well-defined underlying behavior, the output seems unpredictable.

An interesting property of chaotic functions is that very small differences in the initial value can lead to large differences in the result as the formula is repeatedly applied. You can see this in the `chaos` program by entering numbers that differ by only a small amount. Here is the output from a modified program that shows the results for initial values of 0.25 and 0.26 side by side:

```
input   0.25          0.26
---------------------------
      0.731250      0.750360
      0.766441      0.730547
      0.698135      0.767707
      0.821896      0.695499
      0.570894      0.825942
      0.955399      0.560671
      0.166187      0.960644
      0.540418      0.147447
      0.968629      0.490255
      0.118509      0.974630
```

With very similar starting values, the outputs stay similar for a few iterations, but then differ markedly. By about the fifth iteration, there no longer seems to be any relationship between the two models.

These two features of our `chaos` program, apparent unpredictability and extreme sensitivity to initial values, are the hallmarks of chaotic behavior. Chaos has important implications for computer science. It turns out that many phenomena in the real world that we might like to model and predict with our computers exhibit just this kind of chaotic behavior. You may have heard of the so-called *butterfly effect*. Computer models that are used to simulate and predict weather patterns are so sensitive that the effect of a single butterfly flapping its wings in New Jersey might make the difference of whether or not rain is predicted in Peoria.

It's very possible that even with perfect computer modeling, we might never be able to measure existing weather conditions accurately enough to predict weather more than a few days in advance. The measurements simply can't be precise enough to make the predictions accurate over a longer time frame.

As you can see, this small program has a valuable lesson to teach users of computers. As amazing as computers are, the results that they give us are only as useful as the mathematical models on which the programs are based. Computers can give incorrect results because of errors in programs, but even correct programs may produce erroneous results if the models are wrong or the initial inputs are not accurate enough.

1.9 Chapter Summary

This chapter has introduced computers, computer science, and programming. Here is a summary of some of the key concepts:

- A computer is a universal information-processing machine. It can carry out any process that can be described in sufficient detail. A description of the sequence of steps for solving a particular problem is called an algorithm. Algorithms can be turned into software (programs) that determines what the hardware (physical machine) can and does accomplish. The process of creating software is called programming.

- Computer science is the study of what can be computed. Computer scientists use the techniques of design, analysis, and experimentation. Computer science is the foundation of the broader field of computing which includes areas such as networking, databases, and information management systems, to name a few.

- A basic functional view of a computer system comprises a central processing unit (CPU), main memory, secondary memory, and input and output devices. The CPU is the brain of the computer that performs simple arithmetic and logical operations. Information that the CPU acts on (data and programs) is stored in main memory (RAM). More permanent information is stored on secondary memory devices such as magnetic disks, flash memory, and optical devices. Information is entered into the computer via input devices, and output devices display the results.

- Programs are written using a formal notation known as a programming language. There are many different languages, but all share the property of having a precise syntax (form) and semantics (meaning). Computer hardware understands only a very low-level language known as machine language. Programs are usually written using human-oriented, high-level languages such as Python. A high-level language must either be compiled or interpreted in order for the computer to understand it. High-level languages are more portable than machine language.

- Python is an interpreted language. One good way to learn about Python is to use an interactive shell for experimentation. The standard Python distribution includes a program called IDLE that provides a shell as well as facilities for editing Python programs.

- A Python program is a sequence of commands (called statements) for the Python interpreter to execute. Python includes statements to do things such as print output to the screen, get input from the user, calculate the value of a mathematical expression, and perform a sequence of statements multiple times (loop).

- A mathematical model is called chaotic if very small changes in the input lead to large changes in the results, making them seem random or unpredictable. The models of many real-world phenomena exhibit chaotic behavior, which places some limits on the power of computing.

1.10 Exercises

Review Questions

True/False

1. Computer science is the study of computers.
2. The CPU is the “brain” of the computer.
3. Secondary memory is also called RAM.
4. All information that a computer is currently working on is stored in main memory.
5. The syntax of a language is its meaning, and semantics is its form.

6. A function definition is a sequence of statements that defines a new command.
7. A programming environment refers to a place where programmers work.
8. A variable is used to give a name to a value so it can be referred to in other places.
9. A loop is used to skip over a section of a program.
10. A chaotic function can't be computed by a computer.

Multiple Choice

1. What is the fundamental question of computer science?
   a) How fast can a computer compute?
   b) What can be computed?
   c) What is the most effective programming language?
   d) How much money can a programmer make?

2. An algorithm is like a
   a) newspaper b) venus flytrap c) drum d) recipe

3. A problem is intractable when
   a) you cannot reverse its solution
   b) it involves tractors
   c) it has many solutions
   d) it is not practical to solve

4. Which of the following is *not* an example of secondary memory?
   a) RAM b) hard drive c) USB flash drive d) DVD

5. Computer languages designed to be used and understood by humans are
   a) natural languages
   b) high-level computer languages
   c) machine languages
   d) fetch-execute languages

6. A statement is
   a) a translation of machine language
   b) a complete computer command
   c) a precise description of a problem
   d) a section of an algorithm

7. One difference between a compiler and an interpreter is
   a) a compiler is a program
   b) a compiler is used to translate high-level language into machine language
   c) a compiler is no longer needed after a program is translated
   d) a compiler processes source code

8. By convention, the statements of a program are often placed in a function called
   a) import b) main c) program d) IDLE

9. Which of the following is *not* true of comments?
   a) They make a program more efficient.
   b) They are intended for human readers.
   c) They are ignored by Python.
   d) In Python, they begin with a pound sign (#).

10. The items listed in the parentheses of a function definition are called
    a) parentheticals
    b) parameters
    c) arguments
    d) both b) and c) are correct

Discussion

1. Compare and contrast the following pairs of concepts from the chapter:

   a) Hardware vs. Software

   b) Algorithm vs. Program

   c) Programming Language vs. Natural Language

   d) High-Level Language vs. Machine Language

   e) Interpreter vs. Compiler

   f) Syntax vs. Semantics

2. List and explain in your own words the role of each of the five basic functional units of a computer depicted in Figure 1.1.

3. Write a detailed algorithm for making a peanut butter and jelly sandwich (or some other everyday activity). You should assume that you are talking to someone who is conceptually able to do the task, but has never actually done it before. For example, you might be telling a young child.

4. As you will learn in a later chapter, many of the numbers stored in a computer are not exact values, but rather close approximations. For example, the value 0.1 might be stored as 0.10000000000000000555. Usually, such small differences are not a problem; however, given what you have learned about chaotic behavior in Chapter 1, you should realize the need for caution in certain situations. Can you think of examples where this might be a problem? Explain.

5. Trace through the `chaos` program from Section 1.6 by hand using 0.15 as the input value. Show the sequence of output that results.

Programming Exercises

1. Start up an interactive Python session and try typing in each of the following commands. Write down the results you see.

   a) `print("Hello, world!")`

   b) `print("Hello", "world!")`

   c) `print(3)`

   d) `print(3.0)`

   e) `print(2 + 3)`

   f) `print(2.0 + 3.0)`

   g) `print("2" + "3")`

   h) `print("2 + 3 =", 2 + 3)`

   i) `print(2 * 3)`

   j) `print(2 ** 3)`

   k) `print(7 / 3)`

   l) `print(7 // 3)`

2. Enter and run the `chaos` program from Section 1.6. Try it out with various values of input to see that it functions as described in the chapter.

3. Modify the `chaos` program using 2.0 in place of 3.9 as the multiplier in the logistic function. Your modified line of code should look like this:

```
x = 2.0 * x * (1 - x)
```

Run the program for various input values and compare the results to those obtained from the original program. Write a short paragraph describing any differences that you notice in the behavior of the two versions.

4. Modify the `chaos` program so that it prints out 20 values instead of 10.

5. Modify the `chaos` program so that the number of values to print is determined by the user. You will have to add a line near the top of the program to get another value from the user:

   ```
   n = eval(input("How many numbers should I print? "))
   ```

   Then you will need to change the loop to use `n` instead of a specific number.

6. The calculation performed in the `chaos` program can be written in a number of ways that are algebraically equivalent. Write a version of the program for each of the following ways of doing the computation. Have your modified programs print out 100 iterations of the calculation and compare the results when run on the same input.

   a) `3.9 * x * (1 - x)`

   b) `3.9 * (x - x * x)`

   c) `3.9 * x - 3.9 * x * x`

   Explain the results of this experiment. *Hint*: See discussion question number 4, above.

7. (Advanced) Modify the `chaos` program so that it accepts two inputs and then prints a table with two columns similar to the one shown in Section 1.8. (*Note*: You will probably not be able to get the columns to line up as nicely as those in the example. Chapter 5 discusses how to print numbers with a fixed number of decimal places.)

Chapter 2 Writing Simple Programs

Objectives

- To know the steps in an orderly software development process.
- To understand programs following the input, process, output (IPO) pattern and be able to modify them in simple ways.
- To understand the rules for forming valid Python identifiers and expressions.
- To be able to understand and write Python statements to output information to the screen, assign values to variables, get information entered from the keyboard, and perform a counted loop.

2.1 The Software Development Process

As you saw in the previous chapter, it is easy to run programs that have already been written. The harder part is actually coming up with a program in the first place. Computers are very literal, and they must be told what to do right down to the last detail. Writing large programs is a daunting challenge. It would be almost impossible without a systematic approach.

The process of creating a program is often broken down into stages according to the information that is produced in each phase. In a nutshell, here's what you should do:

Analyze the Problem Figure out exactly what the problem to be solved is. Try to understand as much as possible about it. Until you really know what the problem is, you cannot begin to solve it.

Determine Specifications Describe exactly what your program will do. At this point, you should not worry about *how* your program will work, but rather about deciding exactly *what* it will accomplish. For simple programs this involves carefully describing what the inputs and outputs of the program will be and how they relate to each other.

Create a Design Formulate the overall structure of the program. This is where the *how* of the program gets worked out. The main task is to design the algorithm(s) that will meet the specifications.

Implement the Design Translate the design into a computer language and put it into the computer. In this book, we will be implementing our algorithms as Python programs.

Test/Debug the Program Try out your program and see whether it works as expected. If there are any errors (often called *bugs*), then you should go back and fix them. The process of locating and fixing errors is called *debugging* a program. During the debugging phase, your goal is to find errors, so you should try everything you can think of that might "break" the program. It's good to keep in mind the old maxim: "Nothing is foolproof because fools are too ingenious."

Maintain the Program Continue developing the program in response to the needs of your users. Most programs are never really finished; they keep evolving over years of use.

2.2 Example Program: Temperature Converter

Let's go through the steps of the software development process with a simple real-world example involving a fictional computer science student, Susan Computewell.

Susan is spending a year studying in Germany. She has no problems with language, as she is fluent in many languages (including Python). Her problem is that she has a hard time figuring out the temperature in the morning so that she knows how to dress for the day. Susan listens to the weather report each morning, but the temperatures are given in degrees Celsius, and she is used to Fahrenheit.

Fortunately, Susan has an idea to solve the problem. Being a computer science major, she never goes anywhere without her laptop computer. She thinks it might be possible that a computer program could help her out.

Susan begins with an analysis of her problem. In this case, the problem is pretty clear: the radio announcer gives temperatures in degrees Celsius, but Susan only comprehends temperatures that are in degrees Fahrenheit.

Next, Susan considers the specifications of a program that might help her out. What should the input be? She decides that her program will allow her to type in the temperature in degrees Celsius. And the output? The program will display the temperature converted into degrees Fahrenheit. Now she needs to specify the exact relationship of the output to the input.

Susan does some quick figuring. She knows that 0 degrees Celsius (freezing) is equal to 32 degrees Fahrenheit, and 100 Celsius (boiling) is equal to 212 Fahrenheit. With this information, she computes the ratio of Fahrenheit to Celsius degrees as $\frac{212-32}{100-0} = \frac{180}{100} = \frac{9}{5}$. Using F to represent the Fahrenheit temperature and C for Celsius, the conversion formula will have the form $F = \frac{9}{5}C + k$ for some constant k. Plugging in 0 and 32 for C and F, respectively, Susan immediately sees that $k = 32$. So the final formula for the relationship is $F = \frac{9}{5}C + 32$. That seems an adequate specification.

Notice that this describes one of many possible programs that could solve this problem. If Susan had a background in the field of Artificial Intelligence (AI), she might consider writing a program that would actually listen to the radio announcer to get the current temperature using speech recognition algorithms. For output, she might have the computer control a robot that goes to her closet and picks an appropriate outfit based on the converted temperature. This would be a much more ambitious project, to say the least!

Certainly, the robot program would also solve the problem identified in the problem analysis. The purpose of specification is to decide exactly what this particular program will do to solve a problem. Susan knows better than to just dive in and start writing a program without first having a clear idea of what she is trying to build.

Susan is now ready to design an algorithm for her problem. She immediately realizes that this is a simple algorithm that follows a standard pattern: *Input, Process, Output* (IPO). Her program will prompt the user for some input information (the Celsius temperature), process it to produce a Fahrenheit temperature, and then output the result by displaying it on the computer screen.

Susan could write her algorithm down in a computer language. However, the precision required to write it out formally tends to stifle the creative process of developing the algorithm. Instead, she writes her algorithm using *pseudocode*. Pseudocode is just precise English that describes what a program does. It is meant to communicate algorithms without all the extra mental overhead of getting the details right in any particular programming language.

Here is Susan's completed algorithm:

```
Input the temperature in degrees Celsius (call it celsius)
Calculate fahrenheit as (9/5)celsius + 32
Output fahrenheit
```

The next step is to translate this design into a Python program. This is straightforward, as each line of the algorithm turns into a corresponding line of Python code.

```
# convert.py
#     A program to convert Celsius temps to Fahrenheit
# by: Susan Computewell

def main():
    celsius = eval(input("What is the Celsius temperature? "))
    fahrenheit = 9/5 * celsius + 32
    print("The temperature is", fahrenheit, "degrees Fahrenheit.")

main()
```

See if you can figure out what each line of this program does. Don't worry if some parts are a bit confusing. They will be discussed in detail in the next section.

After completing her program, Susan tests it to see how well it works. She uses inputs for which she knows the correct answers. Here is the output from two of her tests:

```
What is the Celsius temperature? 0
The temperature is 32.0 degrees Fahrenheit.

What is the Celsius temperature? 100
The temperature is 212.0 degrees Fahrenheit.
```

You can see that Susan used the values of 0 and 100 to test her program. It looks pretty good, and she is satisfied with her solution. She is especially pleased that no debugging seems necessary (which is very unusual).

2.3 Elements of Programs

Now that you know something about the programming process, you are *almost* ready to start writing programs on your own. Before doing that, though, you need a more complete grounding in the fundamentals of Python. The next few sections will discuss technical details that are essential to writing correct programs. This material can seem a bit tedious, but you will have to master these basics before plunging into more interesting waters.

2.3.1 Names

You have already seen that names are an important part of programming. We give names to modules (e.g., `convert`) and to the functions within modules (e.g., `main`). Variables are used to give names to values (e.g., `celsius` and `fahrenheit`). Technically, all these names are called *identifiers*. Python has some rules about how identifiers are formed. Every identifier must begin with a letter or underscore (the "_" character) which may be followed by any sequence of letters, digits, or underscores. This implies that a single identifier cannot contain any spaces.

According to these rules, all of the following are legal names in Python:

```
x
celsius
spam
spam2
SpamAndEggs
Spam_and_Eggs
```

Identifiers are case-sensitive, so `spam`, `Spam`, `sPam`, and `SPAM` are all different names to Python. For the most part, programmers are free to choose any name that conforms to these rules. Good programmers always try to choose names that describe the thing being named.

One important thing to be aware of is that some identifiers are part of Python itself. These names are called *reserved words* or *keywords* and cannot be used as ordinary identifiers. The complete list of Python keywords is shown in Table 2.1.

`False`	`class`	`finally`	`is`	`return`
`None`	`continue`	`for`	`lambda`	`try`
`True`	`def`	`from`	`nonlocal`	`while`
`and`	`del`	`global`	`not`	`with`
`as`	`elif`	`if`	`or`	`yield`
`assert`	`else`	`import`	`pass`	
`break`	`except`	`in`	`raise`	

Table 2.1: Python keywords

Python also includes quite a number of built-in functions, such as the `print` function that we've already been using. While it's technically legal to (re)use the built-in function-name identifiers for other purposes, it's generally a **very bad** idea to do so. For example, if you redefine the meaning of `print`, then you will no longer be able to print things out. You will also seriously confuse any Python programmers who read your program; they expect `print` to refer to the built-in function. A complete list of the built-in functions can be found in Appendix A.

2.3.2 Expressions

Programs manipulate data. So far, we have seen two different kinds of data in our example programs: numbers and text. We'll examine these different data types in great detail in later chapters. For now, you just need to keep in mind that all data has to be stored on the computer in some digital format, and different types of data are stored in different ways.

The fragments of program code that produce or calculate new data values are called *expressions*. The simplest kind of expression is a *literal*. A literal is used to indicate a specific value. In `chaos.py` you can find the numbers `3.9` and `1`. The `convert.py` program contains `9`, `5`, and `32`. These are all examples of numeric literals, and their meaning is obvious: `32` represents, well, 32 (the number 32).

Our programs also manipulated textual data in some simple ways. Computer scientists refer to textual data as *strings*. You can think of a string as just a sequence of printable characters. A string literal is indicated in Python by enclosing the characters in quotation marks (`""`). If you go back and look at our example programs, you will find a number of string literals such as: `"Hello"` and `"Enter a number between 0 and 1: "`. These literals produce strings containing the quoted characters. Note that the quotes themselves are not part of the string. They are just the mechanism to tell Python to create a string.

The process of turning an expression into an underlying data type is called *evaluation*. When you type an expression into a Python shell, the shell evaluates the expression and prints out a textual representation of the result. Consider this small interaction:

```
>>> 32
32
>>> "Hello"
'Hello'
>>> "32"
'32'
```

Notice that when the shell shows the value of a string, it puts the sequence of characters in single quotes. This is a way of letting us know that the value is actually text, not a number (or other data type). In the last interaction, we see that the expression "32" produces a string, not a number. In this case, Python is actually storing the characters "3" and "2," not a representation of the number 32. If that's confusing right now, don't worry too much about it; it will become clearer when we discuss these data types in later chapters.

A simple identifier can also be an expression. We use identifiers as variables to give names to values. When an identifier appears as an expression, its value is retrieved to provide a result for the expression. Here is an interaction with the Python interpreter that illustrates the use of variables as expressions:

```
>>> x = 5
>>> x
5
>>> print(x)
5
>>> print(spam)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'spam' is not defined
```

First the variable x is assigned the value 5 (using the numeric literal 5). In the second line of interaction, we are asking Python to evaluate the expression x. In response, the Python shell prints out 5, which is the value that was just assigned to x. Of course, we get the same result when we explicitly ask Python to print x using a `print` statement. The last interaction shows what happens when we try to use a variable that has not been assigned a value. Python cannot find a value, so it reports a `NameError`. This says that there is no value with that name. The important lesson here is that a variable must always be assigned a value before it can be used in an expression.

More complex and interesting expressions can be constructed by combining simpler expressions with *operators*. For numbers, Python provides the normal set of mathematical operations: addition, subtraction, multiplication, division, and exponentiation. The corresponding Python operators are `+`, `-`, `*`, `/`, and `**`. Here are some examples of complex expressions from `chaos.py` and `convert.py`:

```
3.9 * x * (1 - x)
9/5 * celsius + 32
```

Spaces are irrelevant within an expression. The last expression could have been written `9/5*celsius+32` and the result would be exactly the same. Usually it's a good idea to place some spaces in expressions to make them easier to read.

Python's mathematical operators obey the same rules of precedence and associativity that you learned in your math classes, including using parentheses to modify the order of evaluation. You should have little trouble constructing complex expressions in your own programs. Do keep in mind that only the round parentheses are allowed in numeric expressions, but you can nest them if necessary to create expressions like this:

```
((x1 - x2) / 2*n) + (spam / k**3)
```

By the way, Python also provides operators for strings. For example, you can "add" strings.

```
>>> "Bat" + "man"
'Batman'
```

This is called *concatenation*. As you can see, the effect is to create a new string that is the result of "gluing" the strings together. You'll see a lot more string operations in Chapter 5.

2.4 Output Statements

Now that you have the basic building blocks, identifier and expression, you are ready for a more complete description of various Python statements. You already know that information can be displayed on screen using Python's built-in function `print`. So far, we have looked at a few examples, but I have not yet explained the `print` function in detail. Like all programming languages, Python has a precise set of rules for the syntax (form) and semantics (meaning) of each statement. Computer scientists have developed sophisticated notations called *meta-languages* for describing programming languages. In this book we will rely on a simple template notation to illustrate the syntax of various statements.

Since `print` is a built-in function, a `print` statement has the same general form as any other function invocation. We type the function name `print` followed by parameters listed in parentheses. Here is how the `print` statement looks using our template notation:

```
print(<expr>, <expr>, ..., <expr>)
print()
```

These two templates show two forms of the `print` statement. The first indicates that a `print` statement can consist of the function name `print` followed by a parenthesized sequence of expressions, which are separated by commas. The angle bracket notation (<>) in the template is used to indicate "slots" that are filled in by other fragments of Python code. The name inside the brackets indicates what is missing; `expr` stands for an expression. The ellipsis ("...") denotes an indefinite series (of expressions, in this case). You don't actually type the dots. The second version of the statement shows that it's also legal to have a `print` without any expressions to print.

As far as semantics is concerned, a `print` statement displays information in textual form. Any supplied expressions are evaluated left to right, and the resulting values are displayed on a line of output in a left-to-right fashion. By default, a single blank space character is placed between the displayed values. As an example, this sequence of `print` statements:

```
print(3+4)
print(3, 4, 3 + 4)
print()
print("The answer is", 3 + 4)
```

produces this output:

```
7
3 4 7

The answer is 7
```

The last statement illustrates how string literal expressions are often used in `print` statements as a convenient way of labeling output.

Notice that successive `print` statements normally display on separate lines of the screen. A bare `print` (no parameters) produces a blank line of output. Underneath, what's really happening is that the `print` function automatically appends some ending text after all of the supplied expressions are printed. By default, that ending text is a special marker character (denoted as `"\n"`) that signals the end of a line. We can modify that behavior by including an additional parameter that explicitly overrides this default. This is done using a special syntax for named or *keyword* parameters.

A template for the `print` statement including the keyword parameter to specify the ending text looks like this:

```
print(<expr>, <expr>, ..., <expr>, end="\n")
```

The keyword for the named parameter is `end` and it is given a value using = notation, similar to variable assignment. Notice in the template I have shown its default value, the end-of-line character. This is a standard way of showing what value a keyword parameter will have when it is not explicitly given some other value.

One common use of the `end` parameter in `print` statements is to allow multiple `print`s to build up a single line of output. For example:

```
print("The answer is", end=" ")
print(3 + 4)
```

produces the single line of output:

```
The answer is 7
```

Notice how the output from the first `print` statement ends with a space (`" "`) rather than an end-of-line character. The output from the second statement appears immediately following the space.

2.5 Assignment Statements

One of the most important kinds of statements in Python is the assignment statement. We've already seen a number of these in our previous examples.

2.5.1 Simple Assignment

The basic assignment statement has this form:

```
<variable> = <expr>
```

Here variable is an identifier and expr is an expression. The semantics of the assignment is that the expression on the right side is evaluated to produce a value, which is then associated with the variable named on the left side.

Here are some of the assignments we've already seen:

```
x = 3.9 * x * (1 - x)
fahrenheit = 9 / 5 * celsius + 32
x = 5
```

A variable can be assigned many times. It always retains the value of the most recent assignment. Here is an interactive Python session that demonstrates the point:

```
>>> myVar = 0
>>> myVar
0
>>> myVar = 7
>>> myVar
7
>>> myVar = myVar + 1
>>> myVar
8
```

The last assignment statement shows how the current value of a variable can be used to update its value. In this case I simply added 1 to the previous value. The chaos.py program from Chapter 1 did something similar, though a bit more complex. Remember, the values of variables can change; that's why they're called variables.

Sometimes it's helpful to think of a variable as a sort of named storage location in computer memory, a box that we can put a value in. When the variable changes, the old value is erased and a new one written in. Figure 2.1 shows how we might picture the effect of x = x + 1 using this model. This is exactly the way assignment works in some computer languages. It's also a very simple way to view the effect of assignment, and you'll find pictures similar to this throughout the book.

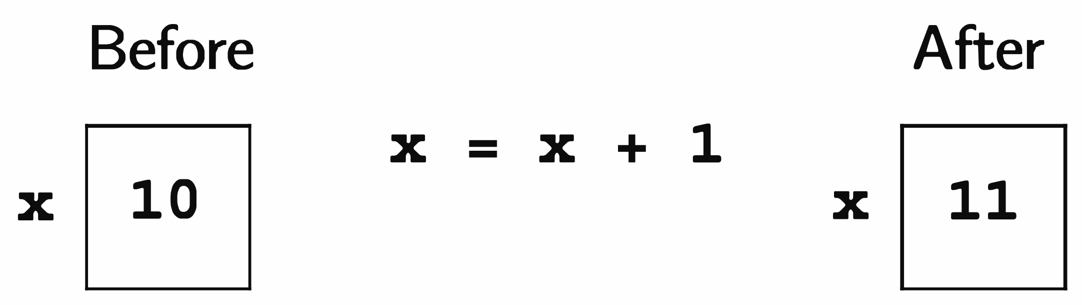


Figure 2.1: Variable as box view of `x = x + 1`

Python assignment statements are actually slightly different from the "variable as a box" model. In Python, values may end up anywhere in memory, and variables are used to refer to them. Assigning a variable is like putting one of those little yellow sticky notes on the value and saying, "this is x." Figure 2.2 gives a more accurate picture of the effect of assignment in Python. An arrow is used to show which value a variable refers to. Notice that the old value doesn't get erased by the new one; the variable simply switches to refer to the new value. The effect is like moving the sticky note from one object to another. This is the way assignment actually works in Python, so you'll see some of these sticky-note style pictures sprinkled throughout the book as well.

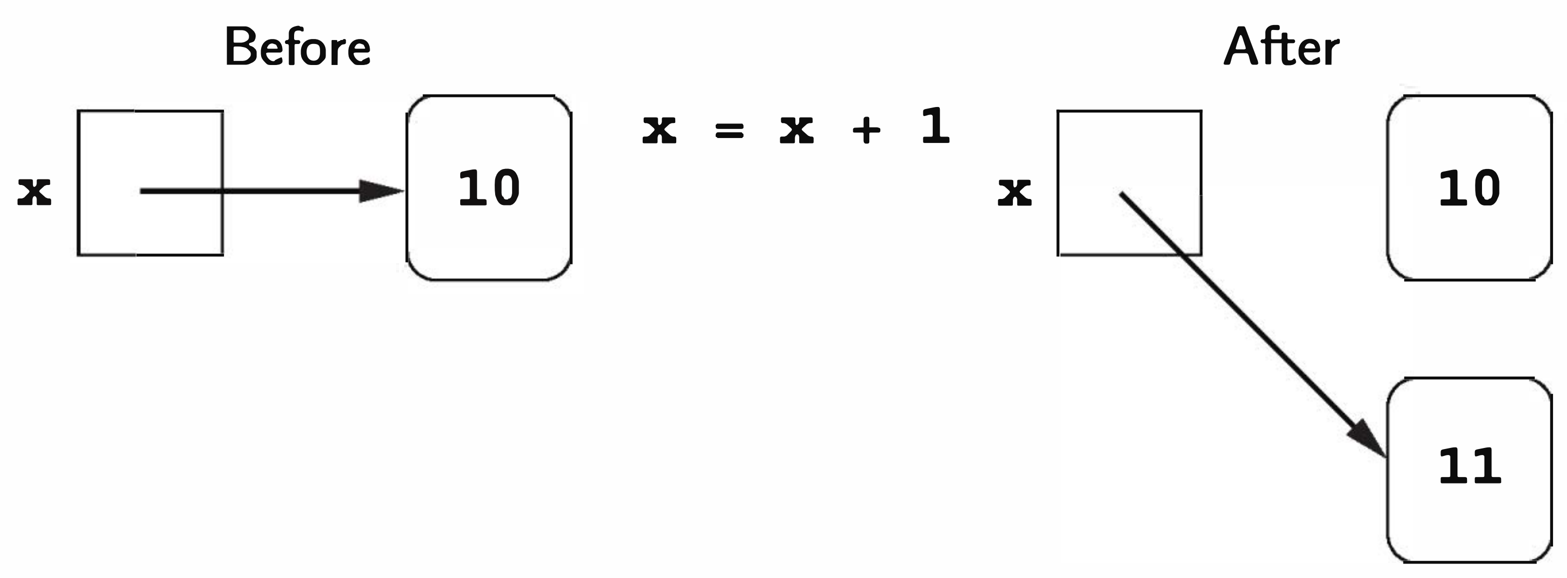


Figure 2.2: Variable as sticky note (Python) view of `x = x + 1`

By the way, even though the assignment statement doesn't directly cause the old value of a variable to be erased and overwritten, you don't have to worry about computer memory getting filled up with the "discarded" values. When a value is no longer referred to by *any* variable, it is no longer useful. Python will automatically clear these values out of memory so that the space can be used for new values. This is like going through your closet and tossing out anything that doesn't have a sticky note to label it. In fact, this process of automatic memory management is actually called *garbage collection*.

2.5.2 Assigning Input

The purpose of an input statement is to get some information from the user of a program and store it into a variable. Some programming languages have a special statement to do this. In Python, input is accomplished using an assignment statement combined with a built-in function called `input`. The exact form of an input statement depends on what type of data you are trying to get from the user. For textual input, the statement will look like this:

```
<variable> = input(<prompt>)
```

Here `<prompt>` is a string expression that is used to prompt the user for input; the prompt is almost always a string literal (i.e., some text inside of quotation marks).

When Python encounters a call to `input`, it prints the prompt on the screen. Python then pauses and waits for the user to type some text and press the `<Enter>` key. Whatever the user types is then stored as a string. Consider this simple interaction:

```
>>> name = input("Enter your name: ")
Enter your name: John Yaya
>>> name
'John Yaya'
```

Executing the `input` statement caused Python to print out the prompt "Enter your name:" and then the interpreter paused waiting for user input. In this example, I typed `John Yaya`. As a result, the string `'John Yaya'` is remembered in the variable `name`. Evaluating `name` gives back the string of characters that I typed.

When the user input is a number, we need a slightly more complicated form of `input` statement:

```
<variable> = eval(input(<prompt>))
```

Here I've added another built-in Python function `eval` that is "wrapped around" the `input` function. As you might guess, `eval` is short for "evaluate." In this form, the text typed by the user is evaluated as an expression to produce the value that is stored into the variable. So, for example, the string `"32"` becomes the number 32. If you look back at the example programs so far, you'll see a couple of examples where we've gotten numbers from the user like this.

```
x = eval(input("Please enter a number between 0 and 1: "))
celsius = eval(input("What is the Celsius temperature? "))
```

The important thing to remember is that you need to `eval` the `input` when you want a number instead of some raw text (a string).

If you are reading the example programs carefully, you probably noticed the blank space inside the quotes at the end of all these prompts. I usually put a space at the end of a prompt so that the input that the user types does not start right next to the prompt. Putting a space in makes the interaction easier to read and understand.

Although our numeric examples specifically prompted the user to enter a number, what the user types in this case is just a numeric literal—a simple Python expression. In fact, any valid expression would be just as acceptable. Consider the following interaction with the Python interpreter:

```
>>> ans = eval(input("Enter an expression: "))
Enter an expression: 3 + 4 * 5
>>> print(ans)
23
>>>
```

Here, when prompted to enter an expression, the user typed "3 + 4 * 5." Python evaluated this expression (via `eval`) and assigned the value to the variable `ans`. When printed, we see that `ans` got the value 23 as expected. In a sense, the `input-eval` combination is like a delayed expression. The example interaction produced exactly the same result as if we had simply written `ans = 3 + 4 * 5`. The difference is that the expression was supplied by the user at the time the statement was executed instead of being typed by the programmer when the program was written.

Beware: the `eval` function is very powerful and *also potentially dangerous*. As this example illustrates, when we evaluate user input, we are essentially allowing the user to enter a portion of our program. Python will dutifully evaluate whatever they type. Someone who knows Python could exploit this ability to enter malicious instructions. For example, the user could type an expression that captures private information or deletes files on the computer. In computer security, this is called a *code injection* attack, because an attacker is injecting malicious code into the running program.

As a beginning programmer writing programs for your own personal use, computer scecurity is not much of an issue; if you are sitting at the computer running a Python program, then you probably have full access to the system and can find much easier ways to, say, delete all your files. However, when the input to a program is coming from untrusted sources, say from users on the Internet, the use of `eval` could be disasterous. Fortunately, you will see some safer alternatives in the next chapter.

2.5.3 Simultaneous Assignment

There is an alternative form of the assignment statement that allows us to calculate several values all at the same time. It looks like this:

```
<var1>, <var2>, ..., <varn> = <expr1>, <expr2>, ..., <exprn>
```

This is called *simultaneous assignment*. Semantically, this tells Python to evaluate all the expressions on the right-hand side and then assign these values to the corresponding variables named on the left-hand side. Here's an example:

```
sum, diff = x+y, x-y
```

Here `sum` would get the sum of `x` and `y`, and `diff` would get the difference.

This form of assignment seems strange at first, but it can prove remarkably useful. Here's an example: Suppose you have two variables `x` and `y`, and you want to swap the values. That is, you want the value currently stored in `x` to be in `y` and the value that is currently in `y` to be stored in `x`. At first, you might think this could be done with two simple assignments:

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

This doesn't work. We can trace the execution of these statements step by step to see why.

Suppose `x` and `y` start with the values 2 and 4. Let's examine the logic of the program to see how the variables change. The following sequence uses comments to describe what happens to the variables as these two statements are executed:

```
# variables       x  y
# initial values  2  4
x = y
```

```
# now           4  4
y = x
# final         4  4
```

See how the first statement clobbers the original value of x by assigning to it the value of y? When we then assign x to y in the second step, we just end up with two copies of the original y value.

One way to make the swap work is to introduce an additional variable that temporarily remembers the original value of x.

```
temp = x
x = y
y = temp
```

Let's walk through this sequence to see how it works.

```
# variables       x  y  temp
# initial values  2  4  no value yet
temp = x
#                 2  4   2
x = y
#                 4  4   2
y = temp
#                 4  2   2
```

As you can see from the final values of x and y, the swap was successful in this case.

This sort of three-way shuffle is common in other programming languages. In Python, the simultaneous assignment statement offers an elegant alternative. Here is a simpler Python equivalent:

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

Because the assignment is simultaneous, it avoids wiping out one of the original values.

Simultaneous assignment can also be used to get multiple numbers from the user in a single input. Consider this program for averaging exam scores:

```
# avg2.py
#   A simple program to average two exam scores
#   Illustrates use of multiple input
```

```
def main():
    print("This program computes the average of two exam scores.")

    score1, score2 = eval(input("Enter two scores separated by a comma: "))
    average = (score1 + score2) / 2

    print("The average of the scores is:", average)

main()
```

The program prompts for two scores separated by a comma. Suppose the user types `86, 92`. The effect of the `input` statement is then the same as if we had done this assignment:

```
score1, score2 = 86, 92
```

We have gotten a value for each of the variables in one fell swoop. This example used just two values, but it could be generalized to any number of inputs.

Of course, we could have just gotten the input from the user with separate input statements:

```
score1 = eval(input("Enter the first score: "))
score2 = eval(input("Enter the second score: "))
```

In some ways this may be better, as the separate prompts are more informative for the user. In this example the decision as to which approach to take is largely a matter of taste. Sometimes getting multiple values in a single `input` provides a more intuitive user interface, so it's a nice technique to have in your toolkit. Just remember that the multiple values trick will not work for string (non-`eval`ed) input; when the user types a comma it will be just another character in the input string. The comma only becomes a separator when the string is subsequently evaluated.

2.6 Definite Loops

You already know that programmers use loops to execute a sequence of statements multiple times in succession. The simplest kind of loop is called a *definite loop*. This is a loop that will execute a definite number of times. That is, at the point in the program when the loop begins, Python knows how many times to go around (or *iterate*) the body of the loop. For example, the `chaos` program in Chapter 1 used a loop that always executed exactly ten times:

```
for i in range(10):
     x = 3.9 * x * (1 - x)
     print(x)
```

This particular loop pattern is called a *counted loop*, and it is built using a Python `for` statement. Before considering this example in detail, let's take a look at what `for` loops are all about.

A Python `for` loop has this general form:

```
for <var> in <sequence>:
    <body>
```

The body of the loop can be any sequence of Python statements. The extent of the body is indicated by its indentation under the loop heading (the `for <var> in <sequence>:` part).

The variable after the keyword `for` is called the *loop index*. It takes on each successive value in the `sequence`, and the statements in the `body` are executed once for each value. Often the `sequence` portion consists of a *list* of values. Lists are a very important concept in Python, and you will learn more about them in upcoming chapters. For now, it's enough to know that you can create a simple list by placing a sequence of expressions in square brackets. Some interactive examples help to illustrate the point:

```
>>> for i in [0, 1, 2, 3]:
        print(i)

0
1
2
3

>>> for odd in [1, 3, 5, 7, 9]:
        print(odd * odd)

1
9
25
```

```
49
81
```

Can you see what is happening in these two examples? The body of the loop is executed using each successive value in the list. The length of the list determines the number of times the loop executes. In the first example, the list contains the four values 0 through 3, and these successive values of `i` are simply printed. In the second example, `odd` takes on the values of the first five odd natural numbers, and the body of the loop prints the squares of these numbers.

Now, let's go back to the example that began this section (from `chaos.py`) Look again at the loop heading:

```
for i in range(10):
```

Comparing this to the template for the `for` loop shows that the last portion, `range(10)`, must be some kind of sequence. It turns out that `range` is a built-in Python function for generating a sequence of numbers "on the fly." You can think of a `range` as a sort of implicit description of a sequence of numbers. To get a handle on what `range` actually does, we can ask Python to turn a range into a plain old list using another built-in function, `list`:

```
>>> list(range(10))    # turns range(10) into an explicit list
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Do you see what is happening here? The expression `range(10)` produces the sequence of numbers 0 through 9. The loop using `range(10)` is equivalent to one using a list of those numbers.

```
for i in [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]:
```

In general, `range(<expr>)` will produce a sequence of numbers that starts with 0 and goes up to, but does not include, the value of `<expr>`. If you think about it, you will see that the value of the expression determines the number of items in the resulting sequence. In `chaos.py` we did not even care what values the loop index variable used (since `i` was not referred to anywhere in the loop body). We just needed a sequence length of 10 to make the body execute 10 times.

As I mentioned above, this pattern is called a *counted loop*, and it is a very common way to use definite loops. When you want to do something in your program a certain number of times, use a `for` loop with a suitable `range`. This is a recurring Python programming idiom that you need to memorize:

```
for <variable> in range(<expr>):
```

The value of the expression determines how many times the loop executes. The name of the index variable doesn't really matter much; programmers often use `i` or `j` as the loop index variable for counted loops. Just be sure to use an identifier that you are not using for any other purpose. Otherwise you might accidentally wipe out a value that you will need later.

The interesting and useful thing about loops is the way that they alter the "flow of control" in a program. Usually we think of computers as executing a series of instructions in strict sequence. Introducing a loop causes Python to go back and do some statements over and over again. Statements like the `for` loop are called *control structures* because they control the execution of other parts of the program.

Some programmers find it helpful to think of control structures in terms of pictures called *flowcharts*. A flowchart is a diagram that uses boxes to represent different parts of a program and arrows between the boxes to show the sequence of events when the program is running. Figure 2.3 depicts the semantics of the `for` loop as a flowchart.

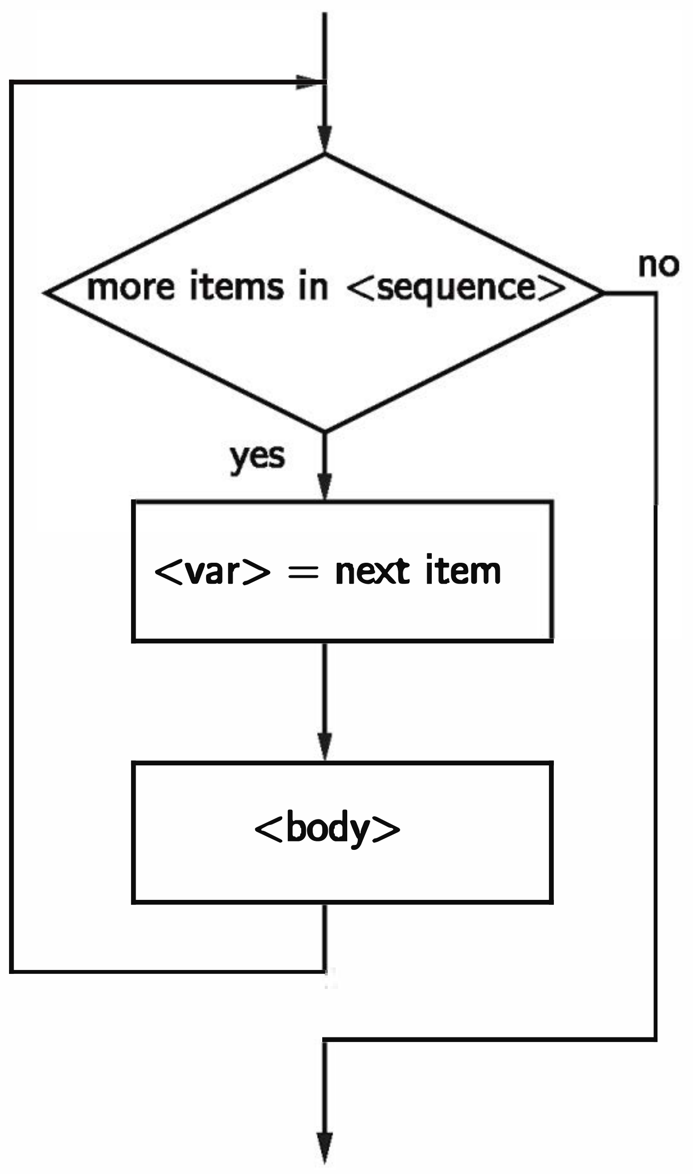


Figure 2.3: Flowchart of a `for` loop

If you are having trouble understanding the `for` loop, you might find it useful to study the flowchart. The diamond-shaped box in the flowchart represents a decision in the program. When Python gets to the loop heading, it checks to see if there are any items left in the sequence. If the answer is "yes," the loop index variable is assigned the next item in the sequence, and then the loop body is executed. Once the body is complete, the program goes back to the loop heading and checks for another value in the sequence. The loop quits when there are no more items, and the program moves on to the statements that come after the loop.

2.7 Example Program: Future Value

Let's close the chapter with one more example of the programming process in action. We want to develop a program to determine the future value of an investment. We'll start with an analysis of the problem. You know that money deposited in a bank account earns interest, and this interest accumulates as the years pass. How much will an account be worth ten years from now? Obviously, it depends on how much money we start with (the principal) and how much interest the account earns. Given the principal and the interest rate, a program should be able to calculate the value of the investment ten years into the future.

We continue by developing the exact specifications for the program. Remember, this is a description of what the program will do. What exactly should the inputs be? We need the user to enter the initial amount to invest, the principal. We will also need some indication of how much interest the account earns. This depends both on the interest rate and how often the interest is compounded. One simple way of handling this is to have the user enter an annual percentage rate. Whatever the actual interest rate and compounding frequency, the annual rate tells us how much the investment accrues in one year. If the annual interest is 3%, then a $100 investment will grow to $103 in one year's time. How should the user represent an annual rate of 3%? There are a number of reasonable choices. Let's assume the user supplies a decimal, so the rate would be entered as 0.03.

This leads us to the following specification:

Program Future Value

Inputs

principal The amount of money being invested in dollars.

APR The annual percentage rate expressed as a decimal number.

Output The value of the investment 10 years into the future.

Relationship Value after one year is given by $principal(1 + apr)$. This formula needs to be applied 10 times.

Next we design an algorithm for the program. We'll use pseudocode, so that we can formulate our ideas without worrying about all the rules of Python. Given our specification, the algorithm seems straightforward.

```
Print an introduction
Input the amount of the principal (principal)
Input the annual percentage rate (apr)
Repeat 10 times:
    principal = principal * (1 + apr)
Output the value of principal
```

If you know a little bit about financial math (or just some basic algebra), you probably realize that the loop in this design is not strictly necessary; there is a formula for calculating future value in a single step using exponentiation. I have used a loop here both to illustrate another counted loop, and also because this version will lend itself to some modifications that are discussed in the programming exercises at the end of the chapter. In any case, this design illustrates that sometimes an algorithmic approach to a calculation can make the mathematics easier. Knowing how to calculate the interest for just one year allows us to calculate any number of years into the future.

Now that we've thought the problem all the way through in pseudocode, it's time to put our new Python knowledge to work and develop a program. Each line of the algorithm translates into a statement of Python:

Print an introduction (`print` statement, Section 2.4)
```
print("This program calculates the future value")
print("of a 10-year investment.")
```

Input the amount of the principal (numeric `input`, Section 2.5.2)
```
principal = eval(input("Enter the initial principal:  "))
```

Input the annual percentage rate (numeric `input`, Section 2.5.2)
```
apr = eval(input("Enter the annual interest rate:  "))
```

Repeat 10 times: (counted loop, Section 2.6)
```
for i in range(10):
```

Calculate principal = principal * (1 + apr) (simple assignment, Section 2.5.1)
```
    principal = principal * (1 + apr)
```

Output the value of the principal (`print` statement, Section 2.4)
```
print("The value in 10 years is:", principal)
```

All of the statement types in this program have been discussed in detail in this chapter. If you have any questions, you should go back and review the relevant descriptions. Notice especially the counted loop pattern is used to apply the interest formula 10 times.

That about wraps it up. Here is the completed program:

```
# futval.py
#    A program to compute the value of an investment
#    carried 10 years into the future

def main():
    print("This program calculates the future value")
    print("of a 10-year investment.")

    principal = eval(input("Enter the initial principal: "))
    apr = eval(input("Enter the annual interest rate: "))

    for i in range(10):
        principal = principal * (1 + apr)

    print("The value in 10 years is:", principal)

main()
```

Notice that I have added a few blank lines to separate the input, processing, and output portions of the program. Strategically placed "white space" can help make your programs more readable.

That's as far as I'm taking this example; I leave the testing and debugging as an exercise for you.

2.8 Chapter Summary

This chapter has covered a lot of ground laying out both the process that is used to develop programs and the details of Python that are necessary to implement simple programs. Here is a quick summary of some of the key points:

- Writing programs requires a systematic approach to problem solving and involves the following steps:
    1. Problem Analysis: Studying the problem to be solved.
    2. Program Specification: Deciding exactly what the program will do.
    3. Design: Writing an algorithm in pseudocode.
    4. Implementation: Translating the design into a programming language.
    5. Testing/Debugging: Finding and fixing errors in the program.
    6. Maintenance: Keeping the program up to date with evolving needs.
- Many simple programs follow the input, process, output (IPO) pattern.
- Programs are composed of statements that are built from identifiers and expressions.
- Identifiers are names; they begin with an underscore or letter which can be followed by a combination of letter, digit, or underscore characters. Identifiers in Python are case-sensitive.
- Expressions are the fragments of a program that produce data. An expression can be composed of the following components:

    literals A literal is a representation of a specific value. For example, `3` is a literal representing the number three.

    variables A variable is an identifier that stores a value.

    operators Operators are used to combine expressions into more complex expressions. For example, in `x + 3 * y` the operators `+` and `*` are used.
- The Python operators for numbers include the usual arithmetic operations of addition (`+`), subtraction (`-`), multiplication (`*`), division (`/`), and exponentiation (`**`).

- The Python output statement `print` displays the values of a series of expressions to the screen.
- In Python, assignment of a value to a variable is indicated using the equal sign (`=`). Using assignment, programs can get input from the keyboard. Python also allows simultaneous assignment, which is useful for getting multiple input values with a single prompt.
- The `eval` function can be used to evaluate user input, but it is a security risk and should not be used with input from unknown or untrusted sources.
- Definite loops are loops that execute a known number of times. The Python `for` statement is a definite loop that iterates through a sequence of values. A Python list is often used in a `for` loop to provide a sequence of values for the loop.
- One important use of a `for` statement is in implementing a counted loop, which is a loop designed specifically for the purpose of repeating some portion of the program a specific number of times. A counted loop in Python is created by using the built-in `range` function to produce a suitably sized sequence of numbers.

2.9 Exercises

Review Questions

True/False

1. The best way to write a program is to immediately type in some code and then debug it until it works.
2. An algorithm can be written without using a programming language.
3. Programs no longer require modification after they are written and debugged.
4. Python identifiers must start with a letter or underscore.
5. Keywords make good variable names.
6. Expressions are built from literals, variables, and operators.

7. In Python, `x = x + 1` is a legal statement.
8. Python does not allow the input of multiple values with a single statement.
9. A counted loop is designed to iterate a specific number of times.
10. In a flowchart, diamonds are used to show statement sequences, and rectangles are used for decision points.

Multiple Choice

1. Which of the following is *not* a step in the software development process?
   a) specification b) testing/Debugging
   c) fee setting d) maintenance
2. What is the correct formula for converting Celsius to Fahrenheit?
   a) $F = 9/5(C) + 32$ b) $F = 5/9(C) - 32$
   c) $F = B^2 - 4AC$ d) $F = \frac{212-32}{100-0}$
3. The process of describing exactly *what* a computer program will do to solve a problem is called
   a) design b) implementation c) programming d) specification
4. Which of the following is *not* a legal identifier?
   a) `spam` b) `spAm` c) `2spam` d) `spam4U`
5. Which of the following are *not* used in expressions?
   a) variables b) statements c) operators d) literals
6. Fragments of code that produce or calculate new data values are called
   a) identifiers b) expressions
   c) productive clauses d) assignment statements
7. Which of the following is *not* a part of the IPO pattern?
   a) input b) program c) process d) output
8. The template `for <variable> in range(<expr>)` describes
   a) a general `for` loop b) an assignment statement
   c) a flowchart d) a counted loop
9. Which of the following is the most accurate model of assignment in Python?
   a) sticky-note b) variable-as-box
   c) simultaneous d) plastic-scale

10. In Python, getting user input is done with a special expression called
    a) `for` b) `read` c) simultaneous assignment d) `input`

Discussion

1. List and describe in your own words the six steps in the software development process.

2. Write out the `chaos.py` program (Section 1.6) and identify the parts of the program as follows:

    - Circle each identifier.
    - Underline each expression.
    - Put a comment at the end of each line indicating the type of statement on that line (output, assignment, input, loop, etc.).

3. Explain the relationships among the concepts: definite loop, `for` loop, and counted loop.

4. Show the output from the following fragments:

    a)
    ```
    for i in range(5):
        print(i * i)
    ```
    b)
    ```
    for d in [3,1,4,1,5]:
        print(d, end=" ")
    ```
    c)
    ```
    for i in range(4):
        print("Hello")
    ```
    d)
    ```
    for i in range(5):
        print(i, 2**i)
    ```

5. Why is it a good idea to first write out an algorithm in pseudocode rather than jumping immediately to Python code?

6. The Python `print` function supports other keyword parameters besides `end`. One of these other keyword parameters is `sep`. What do you think the `sep` parameter does? *Hint*: `sep` is short for separator. Test your idea either by trying it interactively or by consulting the Python documentation.

7. What do you think will happen if the following code is executed?

```
print("start")
for i in range(0):
    print("Hello")
print("end")
```

Look at the flowchart for the `for` statement in this chapter to help you figure this out. Then test your prediction by trying out these lines in a program.

Programming Exercises

1. A user-friendly program should print an introduction that tells the user what the program does. Modify the `convert.py` program (Section 2.2) to print an introduction.

2. On many systems with Python, it is possible to run a program by simply clicking (or double-clicking) on the icon of the program file. If you are able to run the `convert.py` program this way, you may discover another usability issue. The program starts running in a new window, but as soon as the program has finished, the window disappears so that you cannot read the results. Add an `input` statement at the end of the program so that it pauses to give the user a chance to read the results. Something like this should work:

   ```
   input("Press the <Enter> key to quit.")
   ```

3. Modify the `avg2.py` program (Section 2.5.3) to find the average of three exam scores.

4. Modify the `convert.py` program (Section 2.2) with a loop so that it executes 5 times before quitting. Each time through the loop, the program should get another temperature from the user and print the converted value.

5. Modify the `convert.py` program (Section 2.2) so that it computes and prints a table of Celsius temperatures and the Fahrenheit equivalents every 10 degrees from 0°C to 100°C.

6. Modify the `futval.py` program (Section 2.7) so that the number of years for the investment is also a user input. Make sure to change the final message to reflect the correct number of years.

7. Suppose you have an investment plan where you invest a certain fixed amount every year. Modify `futval.py` to compute the total accumulation of your investment. The inputs to the program will be the amount to invest each year, the interest rate, and the number of years for the investment.

8. As an alternative to APR, the interest accrued on an account is often described in terms of a nominal rate and the number of compounding periods. For example, if the interest rate is 3% and the interest is compounded quarterly, the account actually earns $\frac{3}{4}$% interest every 3 months.

   Modify the `futval.py` program to use this method of entering the interest rate. The program should prompt the user for the yearly rate (`rate`) and the number of times that the interest is compounded each year (`periods`). To compute the value in ten years, the program will loop `10 * periods` times and accrue `rate/period` interest on each iteration.

9. Write a program that converts temperatures from Fahrenheit to Celsius.

10. Write a program that converts distances measured in kilometers to miles. One kilometer is approximately 0.62 miles.

11. Write a program to perform a unit conversion of your own choosing. Make sure that the program prints an introduction that explains what it does.

12. Write an interactive Python calculator program. The program should allow the user to type a mathematical expression, and then print the value of the expression. Include a loop so that the user can perform many calculations (say, up to 100). *Note*: To quit early, the user can make the program crash by typing a bad expression or simply closing the window that the calculator program is running in. You'll learn better ways of terminating interactive programs in later chapters.

Chapter 3 Computing with Numbers

Objectives

- To understand the concept of data types.
- To be familiar with the basic numeric data types in Python.
- To understand the fundamental principles of how numbers are represented on a computer.
- To be able to use the Python math library.
- To understand the accumulator program pattern.
- To be able to read and write programs that process numerical data.

3.1 Numeric Data Types

When computers were first developed, they were seen primarily as number crunchers, and that is still an important application. As you have seen, problems that involve mathematical formulas are easy to translate into Python programs. In this chapter, we'll take a closer look at programs designed to perform numerical calculations.

The information that is stored and manipulated by computer programs is generically referred to as *data*. Different kinds of data will be stored and manipulated in different ways. Consider this program to calculate the value of loose change:

```
# change.py
#   A program to calculate the value of some change in dollars

def main():
    print("Change Counter")
    print()
    print("Please enter the count of each coin type.")
    quarters = eval(input("Quarters: "))
    dimes = eval(input("Dimes: "))
    nickels = eval(input("Nickels: "))
    pennies = eval(input("Pennies: "))
    total = quarters * .25 + dimes * .10 + nickels * .05 + pennies * .01
    print()
    print("The total value of your change is", total)

main()
```

Here is an example of the output:

```
Change Counter

Please enter the count of each coin type.
Quarters: 5
Dimes: 3
Nickels: 4
Pennies: 6

The total value of your change is 1.81
```

This program actually manipulates two different kinds of numbers. The values entered by the user (5, 3, 4, 6) are whole numbers; they don't have any fractional part. The values of the coins (.25, .10, .05, .01) are decimal representations of fractions. Inside the computer, whole numbers and numbers that have fractional components are stored differently. Technically, we say that these are two different *data types*.

The data type of an object determines what values it can have and what operations can be performed on it. Whole numbers are represented using the *integer* data type (*int* for short). Values of type int can be positive or negative whole numbers. Numbers that can have fractional parts are represented as *floating-point* (or *float*) values. So how do we tell whether a number is an int or a float? A numeric literal that does not contain a decimal point produces an int value, but a literal that has a decimal point is represented by a float (even if the fractional part is 0).

Python provides a special function called `type` that tells us the data type (or "class") of any value. Here is an interaction with the Python interpreter showing the difference between int and float literals:

```
>>> type(3)
<class 'int'>
>>> type(3.14)
<class 'float'>
>>> type(3.0)
<class 'float'>
>>> myInt = -32
>>> type(myInt)
<class 'int'>
>>> myFloat = 32.0
>>> type(myFloat)
<class 'float'>
```

You may be wondering why there are two different data types for numbers. One reason has to do with program style. Values that represent counts can't be fractional; we can't have $3\frac{1}{2}$ quarters, for example. Using an int value tells the reader of a program that the value *can't* be a fraction. Another reason has to do with the efficiency of various operations. The underlying algorithms that perform computer arithmetic are simpler, and can therefore be faster, for ints than the more general algorithms required for float values. Of course, the hardware implementations of floating-point operations on modern processors are highly optimized and may be just as fast the int operations.

Another difference between ints and floats is that the float type can only represent approximations to real numbers. As we will see, there is a limit to the precision, or accuracy, of the stored values. Since float values are not exact, while ints always are, your general rule of thumb should be: If you don't need fractional values, use an int.

A value's data type determines what operations can be used on it. As we have seen, Python supports the usual mathematical operations on numbers. Table 3.1 summarizes these operations. Actually, this table is somewhat misleading. Since these two types have differing underlying representations, they each have their own set of operations. For example, I have listed a single addition operation, but

operator	operation
+	addition
−	subtraction
*	multiplication
/	float division
**	exponentiation
abs()	absolute value
//	integer division
%	remainder

Table 3.1: Python built-in numeric operations

keep in mind that when addition is performed on floats, the computer hardware performs a floating-point addition, whereas with ints the computer performs an integer addition. Python chooses the appropriate underlying operation (int or float) based on the operands.

Consider the following interaction with Python:

```
>>> 3 + 4
7
>>> 3.0 + 4.0
7.0
>>> 3 * 4
12
>>> 3.0 * 4.0
12.0
>>> 4 ** 3
64
>>> 4.0 ** 3
64.0
>>> 4.0 ** 3.0
64.0
>>> abs(5)
5
>>> abs(-3.5)
3.5
>>>
```

For the most part, operations on floats produce floats, and operations on ints produce ints. Most of the time, we don't even worry about what type of operation is being performed; for example, integer addition produces pretty much the same result as floating-point addition, and we can rely on Python to do the right thing.

In the case of division, however, things get a bit more interesting. As the table shows, Python (as of version 3.0) provides two different operators for division. The usual symbol (/) is used for "regular" division and a double slash (//) is used to indicate integer division. The best way to get a handle on the difference between these two is to try them out.

```
>>> 10 / 3
3.3333333333333335
>>> 10.0 / 3.0
3.3333333333333335
>>> 10 / 5
2.0
>>> 10 // 3
3
>>> 10.0 // 3.0
3.0
>>> 10 % 3
1
>>> 10.0 % 3.0
1.0
```

Notice that the / operator always returns a float. Regular division often produces a fractional result, even though the operands may be ints. Python accommodates this by always returning a floating-point number. Are you surprised that the result of 10/3 has a 5 at the very end? Remember, floating-point values are always approximations. This value is as close as Python can get when representing $3\frac{1}{3}$ as a floating-point number.

To get a division that returns an integer result, you can use the integer division operation //. Integer division always produces an integer. Think of integer division as "gozinta." The expression 10 // 3 produces 3 because three gozinta (goes into) ten three times (with a remainder of one). While the result of integer division is always an integer, the data type of the result depends on the data type of the operands. A float integer-divided by a float produces a float with a 0 fractional component. The last two interactions demonstrate the remainder operation %. The remainder of integer-dividing 10 by 3 is 1. Notice again that the data type of the result depends on the type of the operands.

Depending on your math background, you may not have used the integer division or remainder operations before. The thing to keep in mind is that these two operations are closely related. Integer division tells you how many times one number goes into another, and the remainder tells you how much is left over. Mathematically you could write the idea like this: $a = (a//b)(b) + (a\%b)$.

As an example application, suppose we calculated the value of our loose change in cents (rather than dollars). If I have 383 cents, then I can find the number of whole dollars by computing $383//100 = 3$, and the remaining change is $383\%100 = 83$. Thus, I must have a total of three dollars and 83 cents in change.

By the way, although Python (as of version 3.0) treats regular division and integer division as two separate operators, many other computer languages (and earlier Python versions) just use `/` to signify both. When the operands are ints, `/` means integer division, and when they are floats, it signifies regular division. This is a common source of errors. For example, in our temperature conversion program the formula `9/5 * celsius + 32` would not compute the proper result, since `9/5` would evaluate to `1` using integer division. In these languages, you need to be careful to write this expression as `9.0/5.0 * celsius + 32` so that the proper form of division is used, yielding a fractional result.

3.2 Type Conversions and Rounding

There are situations where a value may need to be converted from one data type into another. You already know that combining an int with an int (usually) produces an int, and combining a float with a float creates another float. But what happens if we write an expression that mixes an int with a float? For example, what should the value of `x` be after this assignment statement?

```
x = 5.0 * 2
```

If this is floating-point multiplication, then the result should be the float value 10.0. If an int multiplication is performed, the result is 10. Before reading ahead for the answer, take a minute to consider how you think Python should handle this situation.

In order to make sense of the expression `5.0 * 2`, Python must either change `5.0` to `5` and perform an int operation or convert `2` to `2.0` and perform a floating-point operation. In general, converting a float to an int is a dangerous step, because some information (the fractional part) will be lost. On the other hand, an int can be safely turned into a float just by adding a fractional part of .0. So in *mixed-typed expressions*, Python will automatically convert ints to floats and perform floating-point operations to produce a float result.

Sometimes we may want to perform a type conversion ourselves. This is called an *explicit* type conversion. Python provides the built-in functions `int` and `float` for these occasions. Here are some interactive examples that illustrate their behavior:

```
>>> int(4.5)
4
>>> int(3.9)
3
>>> float(4)
4.0
>>> float(4.5)
4.5
>>> float(int(3.3))
3.0
>>> int(float(3.3))
3
>>> int(float(3))
3
```

As you can see, converting to an int simply discards the fractional part of a float; the value is truncated, not rounded. If you want a rounded result, you could add 0.5 to the value before using `int()`, assuming the value is positive.

A more general way of rounding off numbers is to use the built-in `round` function, which rounds a number to the nearest whole value.

```
>>> round(3.14)
3
>>> round(3.5)
4
```

Notice that calling `round` like this results in an int value. So a simple call to round is an alternative way of converting a float to an int.

If you want to round a float into another float value, you can do that by supplying a second parameter that specifies the number of digits you want after the decimal point. Here's a little interaction playing around with the value of pi:

```
>>> pi = 3.141592653589793
>>> round(pi, 2)
3.14
>>> round(pi,3)
3.142
```

Notice that when we round the approximation of pi to two or three decimal places, we get a float whose displayed value looks like an exactly rounded result. Remember though, floats are approximations; what we really get is a value that's very close to what we requested. The actual stored value is something like 3.1400000000000001243450..., the closest representable floating-point value to 3.14. Fortunately, Python is smart enough to know that we probably don't want to see all of these digits, so it displays the rounded form. That means when you write a program that rounds off a value to two decimal places and print it out, you'll end up seeing two decimal places, just like you expect. In Chapter 5, we'll see how to get even finer control over how numbers appear when printed; then you'll be able to inspect all of the digits, should you want to.

The type conversion functions `int` and `float` can also be used to convert strings of digits into numbers.

```
>>> int("32")
32
>>> float("32")
32.0
>>> float("9.8")
9.8
```

This is particularly useful as a secure alternative to `eval` for getting numeric data from users. As an example, here is an improved version of the change-counting program that opened the chapter:

```
# change2.py
#   A program to calculate the value of some change in dollars

def main():
    print("Change Counter")
    print()
    print("Please enter the count of each coin type.")
    quarters = int(input("Quarters: "))
    dimes = int(input("Dimes: "))
    nickels = int(input("Nickels: "))
    pennies = int(input("Pennies: "))
    total = .25*quarters + .10*dimes + .05*nickels + .01*pennies
    print()
    print("The total value of your change is", total)

main()
```

Using `int` instead of `eval` in the `input` statements ensures that the user may only enter valid whole numbers. Any illegal (non-int) inputs will cause the program to crash with an error message, thus avoiding the risk of a code injection attack (discussed in Section 2.5.2). A side benefit is that this version of the program emphasizes that the inputs should be whole numbers.

The only downside to using numeric type conversions in place of `eval` is that it does not accommodate simultaneous input (getting multiple values in a single input), as the following example ilustrates:

```
>>> # simultaneous input using eval
>>> x,y = eval(input("Enter (x,y): "))
Enter (x,y): 3,4
>>> x
3
>>> y
4
>>> # does not work with float
>>> x,y = float(input("Enter (x,y): "))
Enter (x,y): 3,4
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: could not convert string to float: '3,4'
```

This is a small price to pay for the added security, and you will learn how to overcome this limitation in Chapter 5. As a matter of good practice, you should use appropriate type conversion functions in place of `eval` wherever possible.

3.3 Using the Math Library

Besides the operations listed in Table 3.1, Python provides many other useful mathematical functions in a special math *library*. A library is just a module that contains some useful definitions. Our next program illustrates the use of this library to compute the roots of quadratic equations.

A quadratic equation has the form $ax^2 + bx + c = 0$. Such an equation has two solutions for the value of x given by the quadratic formula:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Let's write a program that can find the solutions to a quadratic equation. The input to the program will be the values of the coefficients a, b, and c. The outputs are the two values given by the quadratic formula. Here's a program that does the job:

```
# quadratic.py
#    A program that computes the real roots of a quadratic equation.
#    Illustrates use of the math library.
#    Note: This program crashes if the equation has no real roots.

import math  # Makes the math library available.

def main():
    print("This program finds the real solutions to a quadratic")
    print()

    a = float(input("Enter coefficient a: "))
    b = float(input("Enter coefficient b: "))
    c = float(input("Enter coefficient c: "))

    discRoot = math.sqrt(b * b - 4 * a * c)
    root1 = (-b + discRoot) / (2 * a)
    root2 = (-b - discRoot) / (2 * a)

    print()
    print("The solutions are:", root1, root2 )

main()
```

This program makes use of the square root function `sqrt` from the math library module. The line at the top of the program,

```
import math
```

tells Python that we are using the math module. Importing a module makes whatever is defined in it available to the program. To compute $\sqrt{x}$, we use `math.sqrt(x)`. This special dot notation tells Python to use the `sqrt` function that "lives" in the `math` module. In the quadratic program we calculate $\sqrt{b^2 - 4ac}$ with the line

```
discRoot = math.sqrt(b * b - 4 * a * c)
```

Here is how the program looks in action:

```
This program finds the real solutions to a quadratic

Enter coefficient a: 3
Enter coefficient b: 4
Enter coefficient c: -2

The solutions are: 0.38742588672279316 -1.7207592200561266
```

This program is fine as long as the quadratics we try to solve have real solutions. However, some inputs will cause the program to crash. Here's another example run:

```
This program finds the real solutions to a quadratic

Enter coefficient a: 1
Enter coefficient b: 2
Enter coefficient c: 3

Traceback (most recent call last):
  File "quadratic.py", line 21, in ?
    main()
  File "quadratic.py", line 14, in main
    discRoot = math.sqrt(b * b - 4 * a * c)
ValueError: math domain error
```

The problem here is that $b^2 - 4ac < 0$, and the `sqrt` function is unable to compute the square root of a negative number. Python prints a `math domain error`. This is telling us that negative numbers are not in the domain of the `sqrt` function. Right now, we don't have the tools to fix this problem, so we'll just have to assume that the user will give us solvable equations.

Actually, `quadratic.py` did not need to use the math library. We could have taken the square root using exponentiation `**`. (Can you see how?) Using `math.sqrt` is somewhat more efficient, and it allowed me to illustrate the use of the math library. In general, if your program requires a common mathematical function, the math library is the first place to look. Table 3.2 shows some of the other functions that are available in the math library:

Python	mathematics	English
`pi`	π	An approximation of pi.
`e`	e	An approximation of e.
`sqrt(x)`	$\sqrt{x}$	The square root of x.
`sin(x)`	$\sin x$	The sine of x.
`cos(x)`	$\cos x$	The cosine of x.
`tan(x)`	$\tan x$	The tangent of x.
`asin(x)`	$\arcsin x$	The inverse of sine x.
`acos(x)`	$\arccos x$	The inverse of cosine x.
`atan(x)`	$\arctan x$	The inverse of tangent x.
`log(x)`	$\ln x$	The natural (base e) logarithm of x.
`log10(x)`	$\log_{10} x$	The common (base 10) logarithm of x.
`exp(x)`	e^x	The exponential of x.
`ceil(x)`	$\lceil x \rceil$	The smallest whole number >= x.
`floor(x)`	$\lfloor x \rfloor$	The largest whole number <= x.

Table 3.2: Some math library functions

3.4 Accumulating Results: Factorials

Suppose you have a root beer sampler pack containing six different kinds of root beer. Drinking the various flavors in different orders might affect how good they taste. If you wanted to try out every possible ordering, how many different orders would there be? It turns out the answer is a surprisingly large number, 720. Do you know where this number comes from? The value 720 is the *factorial* of 6.

In mathematics, factorials are often denoted with an exclamation point (!). The factorial of a whole number n is defined as $n! = n(n-1)(n-2)\ldots(1)$. This happens to be the number of distinct arrangements for n items. Given six items, we compute $6! = (6)(5)(4)(3)(2)(1) = 720$ possible arrangements.

Let's write a program that will compute the factorial of a number entered by the user. The basic outline of our program follows an input, process, output pattern:

```
Input number to take factorial of, n
Compute factorial of n, fact
Output fact
```

Obviously, the tricky part here is in the second step.

How do we actually compute the factorial? Let's try one by hand to get an idea for the process. In computing the factorial of 6, we first multiply $6(5) = 30$. Then we take that result and do another multiplication: $30(4) = 120$. This result is multiplied by 3: $120(3) = 360$. Finally, this result is multiplied by 2: $360(2) = 720$. According to the definition, we then multiply this result by 1, but that won't change the final value of 720.

Now let's try to think about the algorithm more generally. What is actually going on here? We are doing repeated multiplications, and as we go along, we keep track of the running product. This is a very common algorithmic pattern called an *accumulator*. We build up, or accumulate, a final value piece by piece. To accomplish this in a program, we will use an *accumulator variable* and a loop structure. The general pattern looks like this:

```
Initialize the accumulator variable
Loop until final result is reached
    update the value of accumulator variable
```

Realizing this is the pattern that solves the factorial problem, we just need to fill in the details. We will be accumulating the factorial. Let's keep it in a variable called `fact`. Each time through the loop, we need to multiply `fact` by one of the factors $n, (n-1), \ldots, 1$. It looks like we should use a `for` loop that iterates over this sequence of factors. For example, to compute the factorial of 6, we need a loop that works like this:

```
fact = 1
for factor in [6,5,4,3,2,1]:
    fact = fact * factor
```

Take a minute to trace through the execution of this loop and convince yourself that it works. When the loop body first executes, `fact` has the value `1` and `factor` is `6`. So the new value of `fact` is $1 * 6 = 6$. The next time through the loop, factor will be 5, and fact is updated to $6 * 5 = 30$. The pattern continues for each successive factor until the final result of 720 has been accumulated.

The initial assignment of 1 to fact before the loop is essential to get the loop started. Each time through the loop body (including the first), the current value of fact is used to compute the next value. The initialization ensures that fact has a value on the very first iteration. Whenever you use the accumulator pattern, make sure you include the proper initialization. Forgetting this is a common mistake of beginning programmers.

Of course, there are many other ways we could have written this loop. As you know from math class, multiplication is commutative and associative, so it really doesn't matter what order we do the multiplications in. We could just as easily go the other direction. You might also notice that including 1 in the list of factors is unnecessary, since multiplication by 1 does not change the result. Here is another version that computes the same result:

```
fact = 1
for factor in [2,3,4,5,6]:
    fact = fact * factor
```

Unfortunately, neither of these loops solves the original problem. We have hand-coded the list of factors to compute the factorial of 6. What we really want is a program that can compute the factorial of any given input n. We need some way to generate an appropriate sequence of factors from the value of n.

Luckily, this is quite easy to do using the Python range function. Recall that range(n) produces a sequence of numbers starting with 0 and continuing up to, but not including, n. There are other variations of range that can be used to produce different sequences. With two parameters, range(start,n) produces a sequence that starts with the value start and continues up to, but does not include, n. A third version range(start, n, step) is like the two-parameter version, except that it uses step as the increment between numbers. Here are some examples:

```
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

>>> list(range(5,10))
[5, 6, 7, 8, 9]

>>> list(range(5, 10, 3))
[5, 8]
```

Given our input value n, we have a couple of different range commands that produce an appropriate list of factors for computing the factorial of n. To generate them from smallest to largest (à la our second loop), we could use range(2,n+1). Notice how I used n+1 as the second parameter, since the range will go up to but not include this value. We need the +1 to make sure that n itself is included as the last factor.

Another possibility is to generate the factors in the other direction (à la our first loop) using the three-parameter version of range and a negative step to cause the counting to go backwards: range(n,1,-1). This one produces a list starting with n and counting down (step -1) to, but not including 1.

Here then is one possible version of the factorial program:

```
# factorial.py
#    Program to compute the factorial of a number
#    Illustrates for loop with an accumulator

def main():
    n = int(input("Please enter a whole number: "))
    fact = 1
    for factor in range(n,1,-1):
        fact = fact * factor
    print("The factorial of", n, "is", fact)

main()
```

Of course, there are numerous other ways this program could have been written. I have already mentioned changing the order of factors. Another possibility is to initialize fact to n and then use factors starting at $n-1$ (as long as $n > 0$). You might try out some of these variations and see which one you like best.

3.5 Limitations of Computer Arithmetic

It's sometimes suggested that the reason "!" is used to represent factorials is because the function grows very rapidly. For example, here is what happens if we use our program to find the factorial of 100:

```
Please enter a whole number: 100
The factorial of 100 is 9332621544394415268169923885626670049071596826
43816214685929638952175999932299156089414639761565182862536979208272237
582511852109168640000000000000000000000000
```

That's a pretty big number!

Although recent versions of Python have no difficulty with this calculation, older versions of Python (and modern versions of other languages such as C++ and Java) would not fare as well. For example, here's what happens in several runs of a similar program written using Java:

```
# run 1
Please enter a whole number: 6
The factorial is: 720

# run 2
Please enter a whole number: 12
The factorial is: 479001600

# run 3
Please enter a whole number: 13
The factorial is: 1932053504
```

This looks pretty good; we know that $6! = 720$. A quick check also confirms that $12! = 479001600$. Unfortunately, it turns out that $13! = 6227020800$. It appears that the Java program has given us an incorrect answer!

What is going on here? So far, I have talked about numeric data types as representations of familiar numbers such as integers and decimals (fractions). It is important to keep in mind, however, that computer representations of numbers (the actual data types) do not always behave exactly like the numbers that they stand for.

Remember back in Chapter 1 you learned that the computer's CPU can perform very basic operations such as adding or multiplying two numbers? It would be more precise to say that the CPU can perform basic operations on the computer's internal representation of numbers. The problem in this Java program is that it is representing whole numbers using the computer's underlying int data type and relying on the computer's multiplication operation for ints. Unfortunately, these machine ints are not exactly like mathematical integers. There are infinitely many integers, but only a finite range of ints. Inside the computer, ints are stored in a fixed-sized binary representation. To make sense of all this, we need to look at what's going on at the hardware level.

Computer memory is composed of electrical "switches," each of which can be in one of two possible states, basically on or off. Each switch represents a binary digit or *bit* of information. One bit can encode two possibilities, usually represented with the numerals 0 (for off) and 1 (for on). A sequence of bits can be used to represent more possibilities. With two bits, we can represent four things:

bit 2	bit 1
0	0
0	1
1	0
1	1

Three bits allow us to represent eight different values by adding a 0 or 1 to each of the four two-bit patterns:

bit 3	bit 2	bit 1
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

You can see the pattern here. Each extra bit doubles the number of distinct patterns. In general, n bits can represent 2^n different values.

The number of bits that a particular computer uses to represent an int depends on the design of the CPU. Typical PCs today use 32 or 64 bits. For a 32-bit CPU, that means there are 2^{32} possible values. These values are centered at 0 to represent a range of positive and negative integers. Now $\frac{2^{32}}{2} = 2^{31}$. So the range of integers that can be represented in a 32-bit int value is -2^{31} to $2^{31} - 1$. The reason for the -1 on the high end is to account for the representation of 0 in the top half of the range.

Given this knowledge, let's try to make sense of what's happening in the Java factorial example. If the Java program is relying on a 32-bit int representation, what's the largest number it can store? Python can give us a quick answer:

```
>>> 2**31-1
2147483647
```

Notice that this value (about 2.1 billion) lies between 12! (about 480 million) and 13! (about 6.2 billion). That means the Java program is fine for calculating factorials up to 12, but after that the representation "overflows" and the results are garbage. Now you know exactly why the simple Java program can't compute 13! Of course, that leaves us with another puzzle. Why does the modern Python program seem to work quite well computing with large integers?

At first, you might think that Python uses the float data type to get us around the size limitation of the ints. However, it turns out that floats do not really solve this problem. Here is an example run of a modified factorial program that uses floating-point numbers:

```
Please enter a whole number: 30
The factorial of 30 is  2.6525285981219103e+32
```

Although this program runs just fine, after switching to float, we no longer get an exact answer.

A very large (or very small) floating-point value is printed out using *exponential*, or *scientific*, notation. The `e+32` at the end means that the result is equal to $2.6525285981219103 \times 10^{32}$. You can think of the +32 at the end as a marker that shows where the decimal point should be placed. In this case, it must move 32 places to the right to get the actual value. However, there are only 16 digits to the right of the decimal, so we have "lost" the last 16 digits.

Using a float allows us to represent a much larger *range* of values than a 32-bit int, but the amount of *precision* is still fixed. In fact, a computer stores floating-point numbers as a pair of fixed-length (binary) integers. One integer, called the *mantissa*, represents the string of digits in the value and the second, the *exponent*, keeps track of where the whole part ends and the fractional part begins (where the "binary point" goes). Remember I told you that floats are approximations. Now you can see why. Since the underlying numbers are binary, only fractions that involve powers of 2 can be represented exactly; any other fraction produces an infinitely repeating mantissa. (Just like 1/3 produces an infinitely repeating decimal because 3 is not a power of 10.) When an infinitely long mantissa is truncated to a fixed length for storage, the result is a close approximation. The number of bits used for the mantissa determines how precise the appoximations will be, but there is no getting around the fact that they will be approximations.

Fortunately, Python has a better solution for large, exact values. A Python int is not a fixed size, but expands to accommodate whatever value it holds. The only limit is the amount of memory the computer has available to it. When the value is small, Python can just use the computer's underlying int representation and operations. When the value gets larger, Python automatically converts to a representation using more bits. Of course, in order to perform operations on larger numbers, Python has to break down the operations into smaller units that the computer hardware is able to handle—similar to the way you might do long division by hand. These operations will not be as efficient (they require more steps), but they allow our Python ints to grow to arbitrary size. And that's what allows our simple factorial program to compute some whopping large results. This is a very cool feature of Python.

3.6 Chapter Summary

This chapter has filled in some important details concerning programs that do numerical computations. Here is a quick summary of some key concepts:

- The way a computer represents a particular kind of information is called a data type. The data type of an object determines what values it can have and what operations it supports.

- Python has several different data types for representing numeric values, including int and float.

- Whole numbers are generally represented using the int data type, and fractional values are represented using floats. All of the Python numeric data types support standard, built-in mathematical operations: addition (+), subtraction (-), multiplication (*), division (/), integer division (//), remainder (%), exponentiation (**), and absolute value (`abs(x)`).

- Python automatically converts numbers from one data type to another in certain situations. For example, in a mixed-type expression involving ints and floats, Python first converts the ints into floats and then uses float arithmetic.

- Programs may also explicitly convert one data type into another using the functions `float()`, `int()`, and `round()`. Type conversion functions should generally be used in place of `eval` for handling numeric user inputs.

- Additional mathematical functions are defined in the math library. To use these functions, a program must first import the library.

- Numerical results are often calculated by computing the sum or product of a sequence of values. The loop accumulator programming pattern is useful for this sort of calculation.
- Both ints and floats are represented on the underlying computer using a fixed-length sequence of bits. This imposes certain limits on these representations. Hardware ints must be in the range $-2^{31} \ldots (2^{31} - 1)$ on a 32-bit machine. Floats have a finite amount of precision and cannot represent most numbers exactly.
- Python's int data type may be used to store whole numbers of arbitrary size. Int values are automatically converted to longer representations when they become too large for the underlying hardware int. Calculations involving these long ints are less efficient than those that use only small ints.

3.7 Exercises

Review Questions

True/False

1. Information that is stored and manipulated by computers is called data.
2. Since floating-point numbers are extremely accurate, they should generally be used instead of ints.
3. Operations like addition and subtraction are defined in the math library.
4. The number of possible arrangements of n items is equal to $n!$.
5. The `sqrt` function computes the squirt of a number.
6. The float data type is identical to the mathematical concept of a real number.
7. Computers represent numbers using base-2 (binary) representations.
8. A hardware float can represent a larger range of values than a hardware int.
9. Type conversion functions such as `float` are a safe alternative to `eval` for getting a number as user input.

10. In Python, `4+5` produces the same result type as `4.0+5.0`.

Multiple Choice

1. Which of the following is *not* a built-in Python data type?
   a) int b) float c) rational d) string

2. Which of the following is *not* a built-in operation?
   a) `+` b) `%` c) `abs()` d) `sqrt()`

3. In order to use functions in the math library, a program must include
   a) a comment b) a loop c) an operator d) an import statement

4. The value of 4! is
   a) 9 b) 24 c) 41 d) 120

5. The most appropriate data type for storing the value of pi is
   a) int b) float c) irrational d) string

6. The number of distinct values that can be represented using 5 bits is
   a) 5 b) 10 c) 32 d) 50

7. In a mixed-type expression involving ints and floats, Python will convert
   a) floats to ints b) ints to strings
   c) both floats and ints to strings d) ints to floats

8. Which of the following is not a Python type-conversion function?
   a) `float` b) `round` c) `int` d) `abs`

9. The pattern used to compute factorials is
   a) accumulator b) input, process, output
   c) counted loop d) plaid

10. In modern Python, an int value that grows larger than the underlying hardware int
    a) causes an overflow b) converts to float
    c) breaks the computer d) uses more memory

Discussion

1. Show the result of evaluating each expression. Be sure that the value is in the proper form to indicate its type (int or float). If the expression is illegal, explain why.

a) `4.0 / 10.0 + 3.5 * 2`

b) `10 % 4 + 6 / 2`

b) `abs(4 - 20 // 3) ** 3`

d) `sqrt(4.5 - 5.0) + 7 * 3`

e) `3 * 10 // 3 + 10 % 3`

f) `3 ** 3`

2. Translate each of the following mathematical expressions into an equivalent Python expression. You may assume that the math library has been imported (via `import math`).

a) $(3+4)(5)$

b) $\frac{n(n-1)}{2}$

c) $4\pi r^2$

d) $\sqrt{r(\cos a)^2 + r(\sin b)^2}$

e) $\frac{y2-y1}{x2-x1}$

3. Show the sequence of numbers that would be generated by each of the following `range` expressions.

a) `range(5)`

b) `range(3, 10)`

c) `range(4, 13, 3)`

d) `range(15, 5, -2)`

e) `range(5, 3)`

4. Show the output that would be generated by each of the following program fragments.

a)
```
for i in range(1, 11):
    print(i*i)
```

b)
```
for i in [1,3,5,7,9]:
    print(i, ":", i**3)
print(i)
```

c)
```
x = 2
y = 10
for j in range(0, y, x):
    print(j, end="")
    print(x + y)
print("done")
```

d)
```
ans = 0
for i in range(1, 11):
    ans = ans + i*i
    print(i)
print (ans)
```

5. What do you think will happen if you use a negative number as the second parameter in the `round` function? For example, what should be the result of `round(314.159265, -1)`? Explain the rationale for your answer. After you've written your answer, consult the Python documentation or try out some examples to see what Python actually does in this case.

6. What do you think will happen when the operands to the integer division or remainder operations are negative? Consider each of the following cases and try to predict the result. Then try them out in Python. *Hint*: Recall the magic formula $a = (a//b)(b) + (a\%b)$.

   a) `-10 // 3`
   b) `-10 % 3`
   c) `10 // -3`
   d) `10 % -3`
   e) `-10 // -3`

Programming Exercises

1. Write a program to calculate the volume and surface area of a sphere from its radius, given as input. Here are some formulas that might be useful:

$$V = 4/3\pi r^3$$

$$A = 4\pi r^2$$

2. Write a program that calculates the cost per square inch of a circular pizza, given its diameter and price. The formula for area is $A = \pi r^2$.

3. Write a program that computes the molecular weight of a carbohydrate (in grams per mole) based on the number of hydrogen, carbon, and oxygen atoms in the molecule. The program should prompt the user to enter the number of hydrogen atoms, the number of carbon atoms, and the number of oxygen atoms. The program then prints the total combined molecular weight of all the atoms based on these individual atom weights:

Atom	Weight (grams / mole)
H	1.00794
C	12.0107
O	15.9994

For example, the molecular weight of water (H_2O) is: $2(1.00794) + 15.9994 = 18.01528$.

4. Write a program that determines the distance to a lightning strike based on the time elapsed between the flash and the sound of thunder. The speed of sound is approximately 1100 ft/sec and 1 mile is 5280 ft.

5. The Konditorei coffee shop sells coffee at $10.50 a pound plus the cost of shipping. Each order ships for $0.86 per pound + $1.50 fixed cost for overhead. Write a program that calculates the cost of an order.

6. Two points in a plane are specified using the coordinates (x1,y1) and (x2,y2). Write a program that calculates the slope of a line through two (non-vertical) points entered by the user.

$$slope = \frac{y2 - y1}{x2 - x1}$$

7. Write a program that accepts two points (see previous problem) and determines the distance between them.

$$distance = \sqrt{(x2 - x1)^2 + (y2 - y1)^2}$$

8. The Gregorian epact is the number of days between January 1^{st} and the previous new moon. This value is used to figure out the date of Easter. It is calculated by these formulas (using int arithmetic):

$$C = year//100$$

$$epact = (8 + (C//4) - C + ((8C + 13)//25) + 11(year\%19))\%30$$

Write a program that prompts the user for a 4-digit year and then outputs the value of the epact.

9. Write a program to calculate the area of a triangle given the length of its three sides—a, b, and c—using these formulas:

$$s = \frac{a+b+c}{2}$$

$$A = \sqrt{s(s-a)(s-b)(s-c)}$$

10. Write a program to determine the length of a ladder required to reach a given height when leaned against a house. The height and angle of the ladder are given as inputs. To compute length use:

$$length = \frac{height}{\sin angle}$$

Note: The angle must be in radians. Prompt for an angle in degrees and use this formula to convert:

$$radians = \frac{\pi}{180} degrees$$

11. Write a program to find the sum of the first n natural numbers, where the value of n is provided by the user.

12. Write a program to find the sum of the cubes of the first n natural numbers where the value of n is provided by the user.

13. Write a program to sum a series of numbers entered by the user. The program should first prompt the user for how many numbers are to be summed. The program should then prompt the user for each of the numbers in turn and print out a total sum after all the numbers have been entered. *Hint*: Use an input statement in the body of the loop.

14. Write a program that finds the average of a series of numbers entered by the user. As in the previous problem, the program will first ask the user how many numbers there are. *Note*: The average should always be a float, even if the user inputs are all ints.

15. Write a program that approximates the value of pi by summing the terms of this series: $4/1 - 4/3 + 4/5 - 4/7 + 4/9 - 4/11 + \ldots$ The program should prompt the user for n, the number of terms to sum, and then output the sum of the first n terms of this series. Have your program subtract the approximation from the value of `math.pi` to see how accurate it is.

16. A Fibonacci sequence is a sequence of numbers where each successive number is the sum of the previous two. The classic Fibonacci sequence begins: 1, 1, 2, 3, 5, 8, 13, ... Write a program that computes the nth Fibonacci number where n is a value input by the user. For example, if $n = 6$, then the result is 8.

17. You have seen that the math library contains a function that computes the square root of numbers. In this exercise, you are to write your own algorithm for computing square roots. One way to solve this problem is to use a guess-and-check approach. You first guess what the square root might be, and then see how close your guess is. You can use this information to make another guess and continue guessing until you have found the square root (or a close approximation to it). One particularly good way of making guesses is to use Newton's method. Suppose `x` is the number we want the root of, and `guess` is the current guessed answer. The guess can be improved by using computing the next guess as:

$$\frac{guess + \frac{x}{guess}}{2}$$

    Write a program that implements Newton's method. The program should prompt the user for the value to find the square root of (`x`) and the number of times to improve the guess. Starting with a `guess` value of `x/2`, your program should loop the specified number of times applying Newton's method and report the final value of `guess`. You should also subtract your estimate from the value of `math.sqrt(x)` to show how close it is.

Chapter 4 Objects and Graphics

Objectives

- To understand the concept of objects and how they can be used to simplify programming.
- To become familiar with the various objects available in the graphics library.
- To be able to create objects in programs and call appropriate methods to perform graphical computations.
- To understand the fundamental concepts of computer graphics, especially the role of coordinate systems and coordinate transformations.
- To understand how to work with both mouse- and text-based input in a graphical programming context.
- To be able to write simple interactive graphics programs using the graphics library.

4.1 Overview

So far we have been writing programs that use the built-in Python data types for numbers and strings. We saw that each data type could represent a certain set of values, and each had a set of associated operations. Basically, we viewed the data as passive entities that were manipulated and combined via active operations. This is a traditional way to view computation. To build complex systems, however, it helps to take a richer view of the relationship between data and operations.

Most modern computer programs are built using an *object-oriented* (OO) approach. Object orientation is not easily defined. It encompasses a number of principles for designing and implementing software, principles that we will return to numerous times throughout the course of this book. This chapter provides a basic introduction to object concepts by way of some computer graphics.

Graphical programming is a lot of fun and provides a great vehicle for learning about objects. In the process, you will also learn the principles of computer graphics that underlie many modern computer applications. Most of the applications that you are familiar with probably have a so-called *graphical user interface* (GUI) that provides visual elements like windows, icons (representative pictures), buttons, and menus.

Interactive graphics programming can be very complicated; entire textbooks are devoted to the intricacies of graphics and graphical interfaces. Industrial-strength GUI applications are usually developed using a dedicated graphics programming framework. Python comes with its own standard GUI module called *Tkinter*. As GUI frameworks go, Tkinter is one of the simplest to use, and Python is a great language for developing real-world GUIs. Still, at this point in your programming career, it would be a challenge to learn the intricacies of any GUI framework, and doing so would not contribute much to the main objectives of this chapter, which are to introduce you to objects and the fundamental principles of computer graphics.

To make learning these basic concepts easier, we will use a graphics library (`graphics.py`) specifically written for use with this textbook. This library is a wrapper around Tkinter that makes it more suitable for beginning programmers. It is freely available as a Python module file[1] and you are welcome to use it as you see fit. Eventually, you may want to study the code for the library itself as a stepping stone to learning how to program directly in Tkinter.

4.2 The Object of Objects

The basic idea of object-oriented development is to view a complex system as the interaction of simpler *objects*. The word *objects* is being used here in a specific technical sense. Part of the challenge of OO programming is figuring out the vocabulary. You can think of an OO object as a sort of active data type that

[1]The graphics module is available from this book's support website.

combines both data and operations. To put it simply, objects *know stuff* (they contain data), and they *can do stuff* (they have operations). Objects interact by sending each other messages. A message is simply a request for an object to perform one of its operations.

Consider a simple example. Suppose we want to develop a data processing system for a college or university. We will need to keep track of considerable information. For starters, we must keep records on the students who attend the school. Each student could be represented in the program as an object. A student object would contain certain data such as name, ID number, courses taken, campus address, home address, GPA, etc. Each student object would also be able to respond to certain requests. For example, to send out a mailing, we would need to print an address for each student. This task might be handled by a `printCampusAddress` operation. When a particular student object is sent the `printCampusAddress` message, it prints out its own address. To print out all the addresses, a program would loop through the collection of student objects and send each one in turn the `printCampusAddress` message.

Objects may refer to other objects. In our example, each course in the college might also be represented by an object. Course objects would know things such as who the instructor is, what students are in the course, what the prerequisites are, and when and where the course meets. One example operation might be `addStudent`, which causes a student to be enrolled in the course. The student being enrolled would be represented by the appropriate student object. Instructors would be another kind of object, as well as rooms, and even times. You can see how successive refinement of these ideas could lead to a rather sophisticated model of the information structure of the college.

As a beginning programmer, you're probably not yet ready to tackle a college information system. For now, we'll study objects in the context of some simple graphics programming.

4.3 Simple Graphics Programming

In order to run the graphical programs and examples in this chapter (and the rest of the book), you will need a copy of the file `graphics.py` that is supplied with the supplemental materials. Using the graphics library is as easy as placing a copy of the `graphics.py` file in the same folder as your graphics program(s). Alternatively, you can place it in a system directory where other Python libraries are stored so that it can be used from any folder on the system.

The graphics library makes it easy to experiment with graphics interactively and write simple graphics programs. As you do, you will be learning principles of object-oriented programming and computer graphics that can be applied in more sophisticated graphical programming environments. The details of the `graphics` module will be explored in later sections. Here we'll concentrate on a basic hands-on introduction to whet your appetite.

As usual, the best way to start learning new concepts is to roll up your sleeves and try out some examples. The first step is to import the graphics module. Assuming you have placed `graphics.py` in an appropriate place, you can import the graphics commands into an interactive Python session. If you are using IDLE, you may have to first "point" IDLE to the folder where you saved `graphics.py`. A simple way to do this is to load and run one of your existing programs from that folder. Then you should be able to import `graphics` into the shell window:

```
>>> import graphics
>>>
```

If this import fails, it means that Python couldn't find the graphics module. Make sure the file is in the correct folder and try again.

Next we need to create a place on the screen where the graphics will appear. That place is a *graphics window* or `GraphWin`, which is provided by `graphics`:

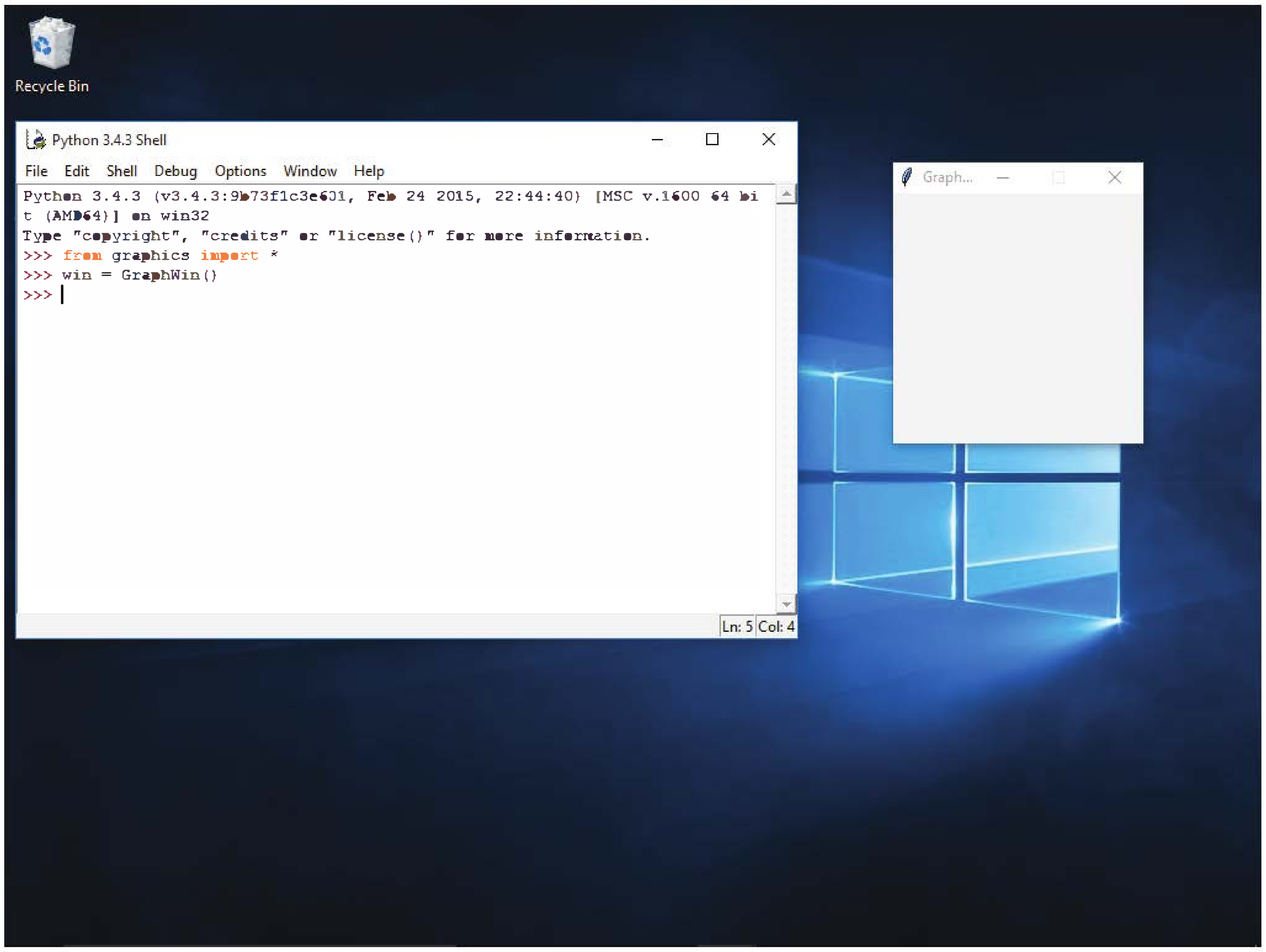
```
>>> win = graphics.GraphWin()
>>>
```

Notice the use of dot notation to invoke the `GraphWin` function that "lives in" the graphics library. This is analogous to when we used `math.sqrt(x)` to invoke the square root function from the math library module. The `GraphWin()` function creates a new window on the screen. The window will have the title "Graphics Window." The `GraphWin` may overlap your Python shell window, so you might have to resize or move the shell to make both windows fully visible. Figure 4.1 shows an example screen view.

The `GraphWin` is an object, and we have assigned it to the variable called `win`. We can now manipulate the window object through this variable. For example, when we are finished with a window, we can destroy it. This is done by issuing the `close` command:

```
>>> win.close()
>>>
```

Typing this command causes the window to vanish from the screen.



```
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 24 2015, 22:44:40) [MSC v.1600 64 bi
t (AMD64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> from graphics import *
>>> win = GraphWin()
>>> 
```

Figure 4.1: Screen shot with a Python shell and a `GraphWin`

Notice that we are again using the dot notation, but now we are using it with a variable name, not a module name, on the left side of the dot. Recall that `win` was earlier assigned as an object of type `GraphWin`. One of the things a `GraphWin` object can do is to close itself. You can think of this command as invoking the `close` operation that is associated with this particular window. The result is that the window disappears from the screen.

By the way, I should mention here that trying out graphics commands interactively like this may be tricky in some environments. If you are using a shell within an IDE such as IDLE, it is possible that on your particular platform a graphics window appears nonresponsive. For example, you may see a "busy" cursor when you mouse over the window, and you may not be able to drag the window to position it. In some cases, your graphics window might be completely hidden underneath the IDE and you have to go searching for it. These glitches are due to the IDE and the graphics window both striving to be in control of your interactions. Regardless of any difficulties you might have playing with the graphics interatively, rest assured that your programs making use of the graphics library should run just fine in most standard environments. They will definitely work under Windows, macOS, and Linux.

We will be using quite a few commands from the graphics library, and it gets tedious having to type the "`graphics.`" notation every time we use one. Python has an alternative form of `import` that can help out:

```
from graphics import *
```

The `from` statement allows you to load specific definitions from a library module. You can either list the names of definitions to be imported or use an asterisk, as shown, to import everything defined in the module. The imported commands become directly available without having to preface them with the module name. After doing this import, we can create a `GraphWin` more simply:

```
win = GraphWin()
```

All of the rest of the graphics examples will assume that the entire graphics module has been imported using `from`.

Let's try our hand at some drawing. A graphics window is actually a collection of tiny points called *pixels* (short for "picture elements"). By controlling the color of each pixel, we control what is displayed in the window. By default, a `GraphWin` is 200 pixels tall and 200 pixels wide. That means there are 40,000 pixels in the `GraphWin`. Drawing a picture by assigning a color to each individual pixel would be a daunting challenge. Instead, we will rely on a library of graphical objects. Each type of object does its own bookkeeping and knows how to draw itself into a `GraphWin`.

The simplest object in the `graphics` module is a `Point`. In geometry, a point is a location in space. A point is located by reference to a coordinate system. Our graphics object `Point` is similar; it can represent a location in a `GraphWin`. We define a point by supplying x and y coordinates (x, y). The x value represents the horizontal location of the point, and the y value represents the vertical.

Traditionally, graphics programmers locate the point $(0, 0)$ in the upper-left corner of the window. Thus x values increase from left to right, and y values increase from top to bottom. In the default 200 x 200 `GraphWin`, the lower-right corner has the coordinates $(199, 199)$. Drawing a `Point` sets the color of the corresponding pixel in the `GraphWin`. The default color for drawing is black.

Here is a sample interaction with Python illustrating the use of `Points`:

```
>>> p = Point(50,60)
>>> p.getX()
50
>>> p.getY()
60
>>> win = GraphWin()
>>> p.draw(win)
>>> p2 = Point(140,100)
>>> p2.draw(win)
```

The first line creates a `Point` located at $(100, 120)$. After the `Point` has been created, its coordinate values can be accessed by the operations `getX` and `getY`. As with all function calls, make sure to put the parentheses on the end when you are attempting to use the operations. A `Point` is drawn into a window using the `draw` operation. In this example, two different `Point` objects (`p` and `p2`) are created and drawn into the `GraphWin` called `win`. Figure 4.2 shows the resulting graphical output.

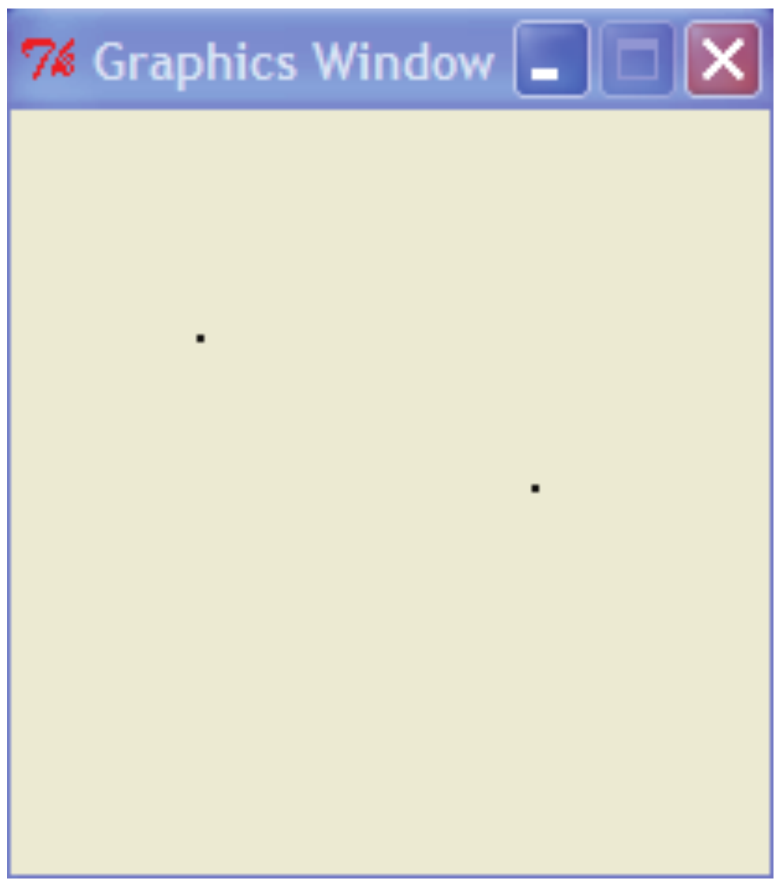


Figure 4.2: Graphics window with two points drawn

In addition to points, the graphics library contains commands for drawing lines, circles, rectangles, ovals, polygons and text. Each of these objects is created and drawn in a similar fashion. Here is a sample interaction to draw various shapes into a `GraphWin`:

```
>>> #### Open a graphics window
>>> win = GraphWin('Shapes')
>>> #### Draw a red circle centered at point (100,100) with radius 30
>>> center = Point(100,100)
>>> circ = Circle(center, 30)
>>> circ.setFill('red')
>>> circ.draw(win)
>>> #### Put a textual label in the center of the circle
>>> label = Text(center, "Red Circle")
>>> label.draw(win)
>>> #### Draw a square using a Rectangle object
>>> rect = Rectangle(Point(30,30), Point(70,70))
>>> rect.draw(win)
>>> #### Draw a line segment using a Line object
>>> line = Line(Point(20,30), Point(180, 165))
>>> line.draw(win)
>>> #### Draw an oval using the Oval object
>>> oval = Oval(Point(20,150),  Point(180,199))
>>> oval.draw(win)
```

Try to figure out what each of these statements does. If you type them in as shown, the final result will look like Figure 4.3.

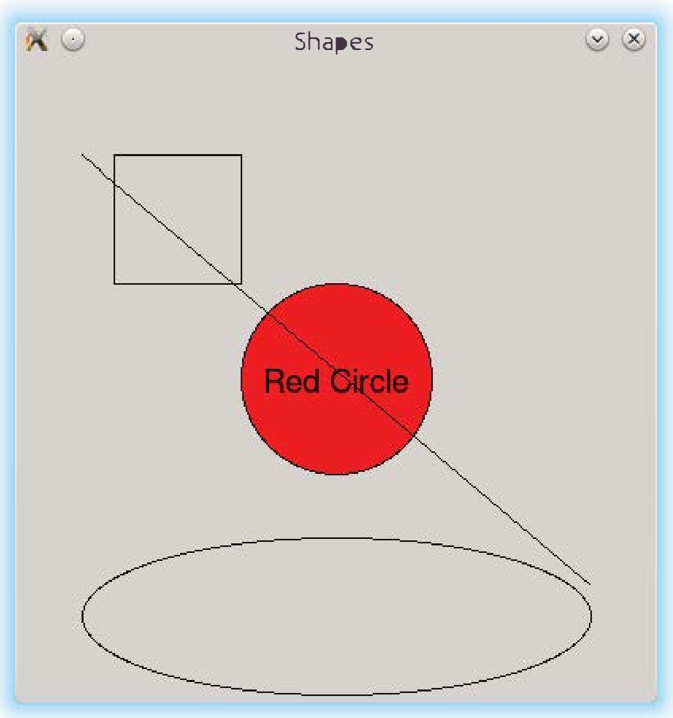


Figure 4.3: Various shapes from the `graphics` module

4.4 Using Graphical Objects

Some of the examples in the above interactions may look a bit strange to you. To really understand the graphics module, we need to take an object-oriented point of view. Remember, objects combine data with operations. Computation is performed by asking an object to carry out one of its operations. In order to make use of objects, you need to know how to create them and how to request operations.

In the interactive examples above, we manipulated several different kinds of objects: `GraphWin`, `Point`, `Circle`, `Oval`, `Line`, `Text`, and `Rectangle`. These are examples of *classes*. Every object is an *instance* of some class, and the class describes the properties the instance will have.

Borrowing a biological metaphor, when we say that Fido is a dog, we are actually saying that Fido is a specific individual in the larger class of all dogs. In OO terminology, Fido is an instance of the dog class. Because Fido is an instance of this class, we expect certain things. Fido has four legs, a tail, a cold, wet nose, and he barks. If Rex is a dog, we expect that he will have similar properties, even though Fido and Rex may differ in specific details such as size or color.

The same ideas hold for our computational objects. We can create two separate instances of `Point`, say `p` and `p2`. Each of these points has an x and y value, and they both support the same set of operations like `getX` and `draw`. These properties hold because the objects are `Points`. However, different instances can vary in specific details such as the values of their coordinates.

To create a new instance of a class, we use a special operation called a *constructor*. A call to a constructor is an expression that creates a brand new object. The general form is as follows:

```
<class-name>(<param1>, <param2>, ...)
```

Here `<class-name>` is the name of the class that we want to create a new instance of, e.g., `Circle` or `Point`. The expressions in the parentheses are any parameters that are required to initialize the object. The number and type of the parameters depends on the class. A `Point` requires two numeric values, while a `GraphWin` can be constructed without any parameters. Often, a constructor is used on the right side of an assignment statement, and the resulting object is immediately assigned to a variable on the left side that is then used to manipulate the object.

To take a concrete example, let's look at what happens when we create a graphical point. Here is a constructor statement from the interactive example above:

```
p = Point(50,60)
```

The constructor for the `Point` class requires two parameters giving the x and y coordinates for the new point. These values are stored as *instance variables* inside the object. In this case, Python creates an instance of `Point` having an x value of 50 and a y value of 60. The resulting point is then assigned to the variable `p`.

A conceptual diagram of the result is shown in Figure 4.4. Note that in this diagram as well as similar ones later on, only the most salient details are shown. `Points` also contain other information such as their color and which window (if any) they are drawn in. Most of this information is set to default values when the `Point` is created.

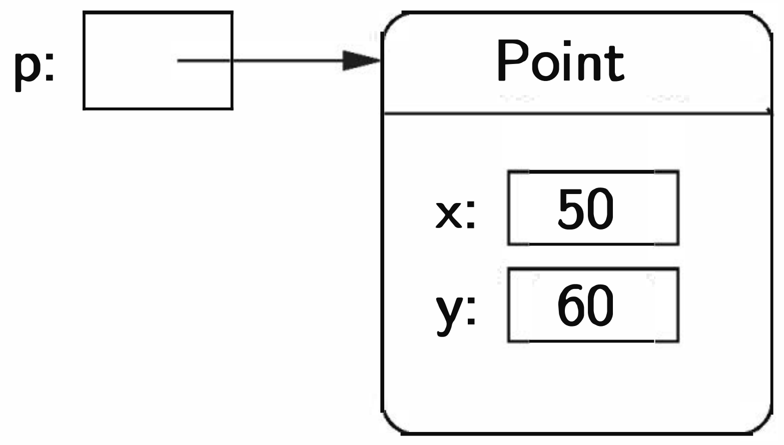


Figure 4.4: The variable `p` refers to a new `Point`

To perform an operation on an object, we send the object a message. The set of messages that an object responds to are called the *methods* of the object. You can think of methods as functions that live inside the object. A method is invoked using dot-notation.

```
<object>.<method-name>(<param1>, <param2>, ...)
```

The number and type of the parameters is determined by the method being used. Some methods require no parameters at all. You can find numerous examples of method invocation in the interactive examples above.

As examples of parameterless methods, consider these two expressions:

```
p.getX()
p.getY()
```

The `getX` and `getY` methods return the x and y values of a point, respectively. Methods such as these are sometimes called *accessors*, because they allow us to access information from the instance variables of the object.

Other methods change the values of an object's instance variables, hence changing the *state* of the object. All of the graphical objects have a `move` method. Here is a specification:

`move(dx,dy)`: Moves the object `dx` units in the x direction and `dy` units in the y direction.

To move the point `p` to the right 10 units, we could use this statement:

```
p.move(10,0)
```

This changes the x instance variable of `p` by adding 10 units. If the point is currently drawn in a `GraphWin`, `move` will also take care of erasing the old image and drawing it in its new position. Methods that change the state of an object are sometimes called *mutators*.

The `move` method must be supplied with two simple numeric parameters indicating the distance to move the object along each dimension. Some methods require parameters that are themselves complex objects. For example, drawing a `Circle` into a `GraphWin` involves two objects. Let's examine a sequence of commands that does this:

```
circ = Circle(Point(100,100), 30)
win = GraphWin()
circ.draw(win)
```

The first line creates a `Circle` with a center located at the `Point` $(100, 100)$ and a radius of 30. Notice that we used the `Point` constructor to create a location for the first parameter to the `Circle` constructor. The second line creates a `GraphWin`. Do you see what is happening in the third line? This is a request for the `Circle` object `circ` to draw itself into the `GraphWin` object `win`. The visible effect of this statement is a circle in the `GraphWin` centered at $(100, 100)$ and having a radius of 30. Behind the scenes, a lot more is happening.

Remember, the `draw` method lives inside the `circ` object. Using information about the center and radius of the circle from the instance variables, the `draw` method issues an appropriate sequence of low-level drawing commands (a sequence of method invocations) to the `GraphWin`. A conceptual picture of the interactions among the `Point`, `Circle` and `GraphWin` objects is shown in Figure 4.5. Fortunately, we don't usually have to worry about these kinds of details; they're all taken care of by the graphical objects. We just create objects, call the appropriate methods, and let them do the work. That's the power of object-oriented programming.

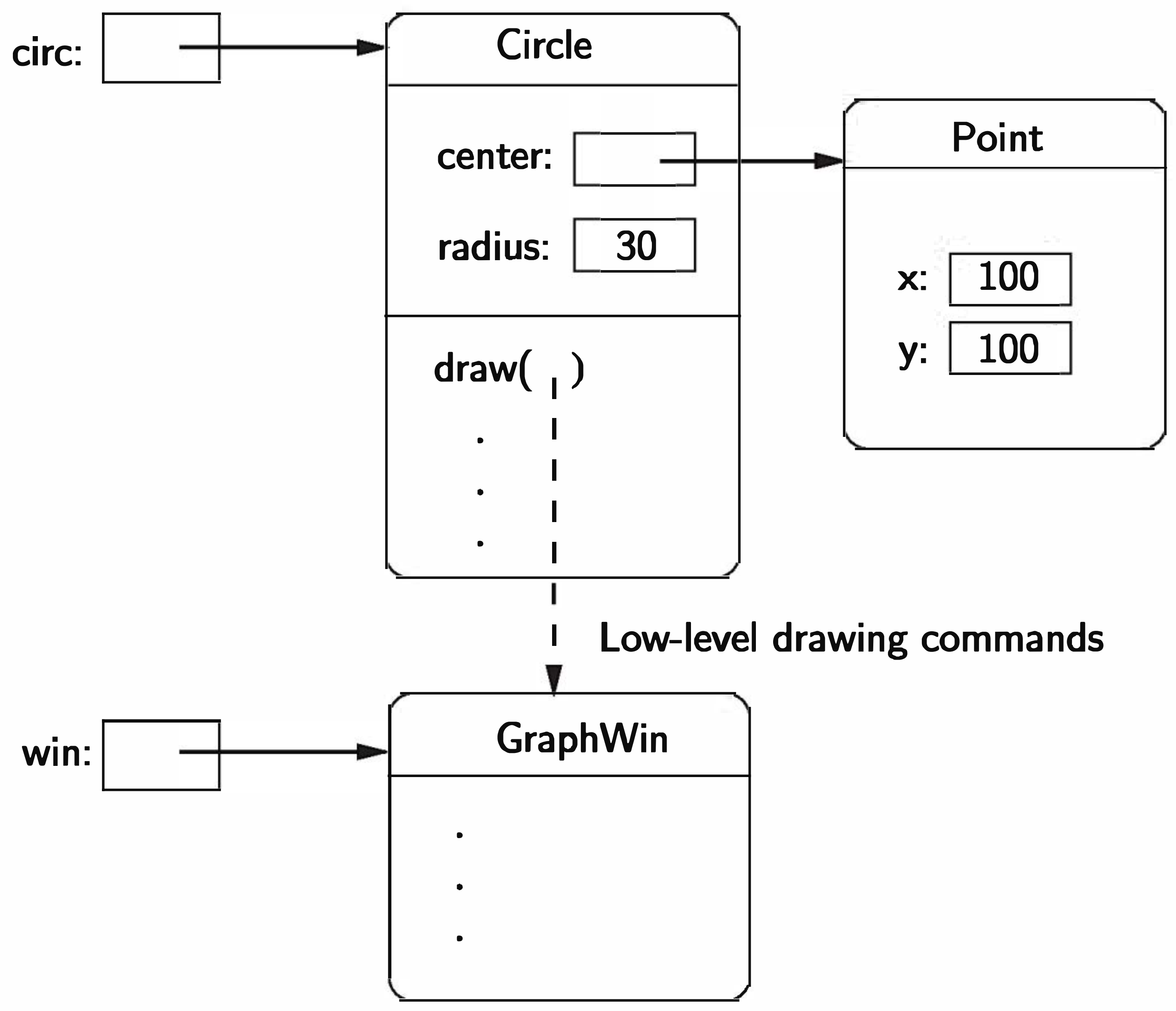


Figure 4.5: Object interactions to draw a circle

There is one subtle “gotcha” that you need to keep in mind when using objects. It is possible for two different variables to refer to exactly the same object; changes made to the object through one variable will also be visible to the other. Suppose, for example, we are trying to write a sequence of code that draws a smiley face. We want to create two eyes that are 20 units apart. Here is a sequence of code intended to draw the eyes:

```
## Incorrect way to create two circles.
leftEye = Circle(Point(80, 50), 5)
leftEye.setFill('yellow')
leftEye.setOutline('red')
rightEye = leftEye
rightEye.move(20,0)
```

The basic idea is to create the left eye and then copy that into a right eye, which is then moved over 20 units.

This doesn't work. The problem here is that only one `Circle` object is created. The assignment

```
rightEye = leftEye
```

simply makes rightEye refer to the very same circle as leftEye. Figure 4.6 shows the situation. When the Circle is moved in the last line of code, both rightEye and leftEye refer to it in its new location on the right side. This situation where two variables refer to the same object is called *aliasing*, and it can sometimes produce rather unexpected results.

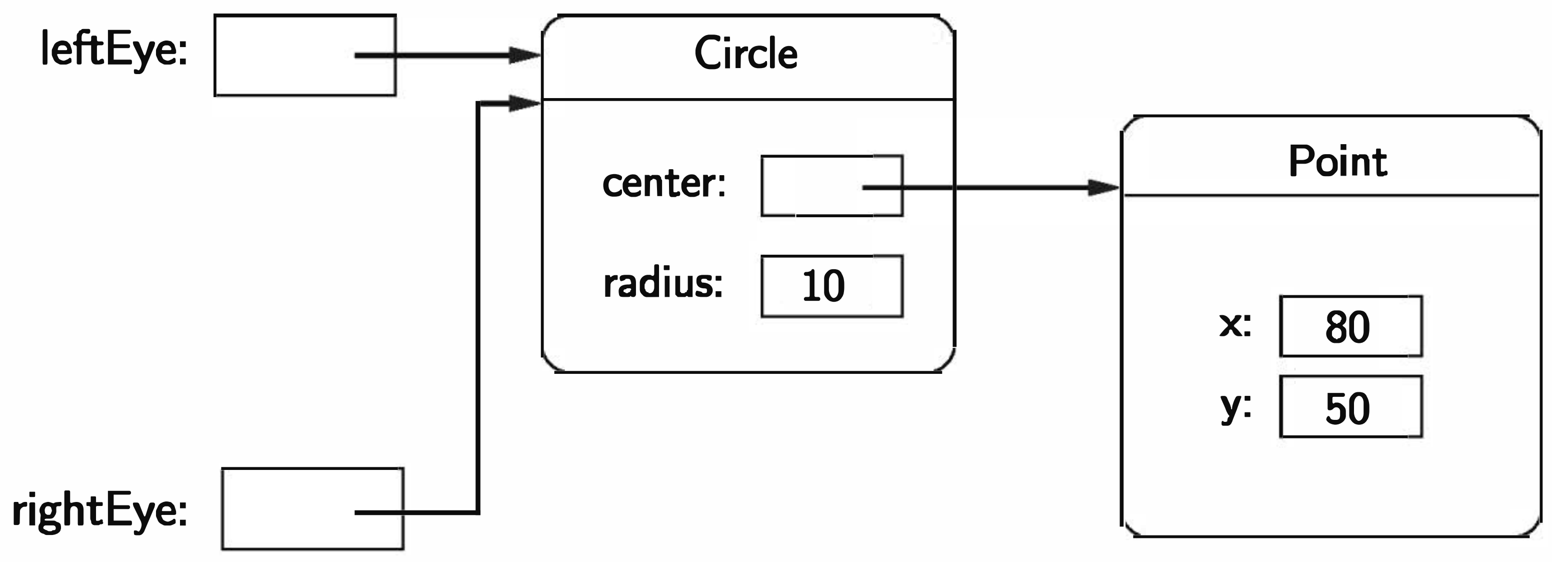


Figure 4.6: Variables leftEye and rightEye are aliases

One solution to this problem would be to create a separate circle for each eye:

```
## A correct way to create two circles.
leftEye = Circle(Point(80, 50), 5)
leftEye.setFill('yellow')
leftEye.setOutline('red')
rightEye = Circle(Point(100, 50), 5)
rightEye.setFill('yellow')
rightEye.setOutline('red')
```

This will certainly work, but it's cumbersome. We had to write duplicated code for the two eyes. That's easy to do using a "cut and paste" approach, but it's not very elegant. If we decide to change the appearance of the eyes, we will have to be sure to make the changes in two places.

The graphics library provides a better solution; all graphical objects support a clone method that makes a copy of the object. Using clone, we can rescue the original approach:

```
## Correct way to create two circles, using clone.
leftEye = Circle(Point(80, 50), 5)
leftEye.setFill('yellow')
leftEye.setOutline('red')
```

```
rightEye = leftEye.clone() # rightEye is an exact copy of the left
rightEye.move(20,0)
```

Strategic use of cloning can make some graphics tasks much easier.

4.5 Graphing Future Value

Now that you have some idea of how to use objects from `graphics`, we're ready to try some real graphics programming. One of the most important uses of graphics is providing a visual representation of data. They say a picture is worth a thousand words; it is almost certainly better than a thousand numbers. Just about any program that manipulates numeric data can be improved with a bit of graphical output. Remember the program in Chapter 2 that computed the future value of a ten-year investment? Let's try our hand at creating a graphical summary.

Programming with graphics requires careful planning. You'll probably want pencil and paper handy to draw some diagrams and scratch out calculations as we go along. As usual, we begin by considering the specification of exactly *what* the program will do.

The original program `futval.py` had two inputs: the amount of money to be invested and the annualized rate of interest. Using these inputs, the program calculated the change in principal year by year for ten years using the formula $\text{principal} = \text{principal} * (1 + \text{apr})$. It then printed out the final value of the principal. In the graphical version, the output will be a ten-year bar graph where the height of successive bars represents the value of the principal in successive years.

Let's use a concrete example for illustration. Suppose we invest $2000 at 10% interest. Table 4.1 shows the growth of the investment over a ten-year period. Our program will display this information in a bar graph. Figure 4.7 shows the same data in graphical form. The graph contains eleven bars. The first bar shows the original value of the principal. For reference, let's number these bars according to the number of years of interest accrued, 0–10.

Here is a rough design for the program:

```
Print an introduction
Get value of principal and apr from user
Create a GraphWin
Draw scale labels on left side of window
Draw bar at position 0 with height corresponding to principal
```

years	value
0	$2,000.00
1	$2,200.00
2	$2,420.00
3	$2,662.00
4	$2,928.20
5	$3,221.02
6	$3,542.12
7	$3,897.43
8	$4,287.18
9	$4,715.90
10	$5,187.49

Table 4.1: Table showing growth of $2000 at 10% interest

```
For successive years 1 through 10
    Calculate principal = principal * (1 + apr)
    Draw a bar for this year having a height corresponding to principal
Wait for user to press Enter.
```

The pause created by the last step is necessary to keep the graphics window displayed so that we can interpret the results. Without such a pause, the program would end, and the `GraphWin` would vanish with it.

While this design gives us the broad brush strokes for our algorithm, there are some very important details that have been glossed over. We must decide exactly how big the graphics window will be and how we will position the objects that appear in this window. For example, what does it mean to draw, say, a bar for year five with height corresponding to $3221.02?

Let's start with the size of the `GraphWin`. Recall that the size of a window is given in terms of the number of pixels in each dimension. Computer screens are also measured in terms of pixels. The number of pixels or *resolution* of the screen is determined by the monitor and graphics card in the computer you use. The lowest resolution screen you are likely to encounter on a personal computer these days is a so-called *extended VGA* screen that is 1024x768 pixels. Most screens are considerably larger. Our default 200x200 pixel window will probably seem a bit small. Let's make the `GraphWin` 320x240; that will make it about 1/8 the size of a small screen.

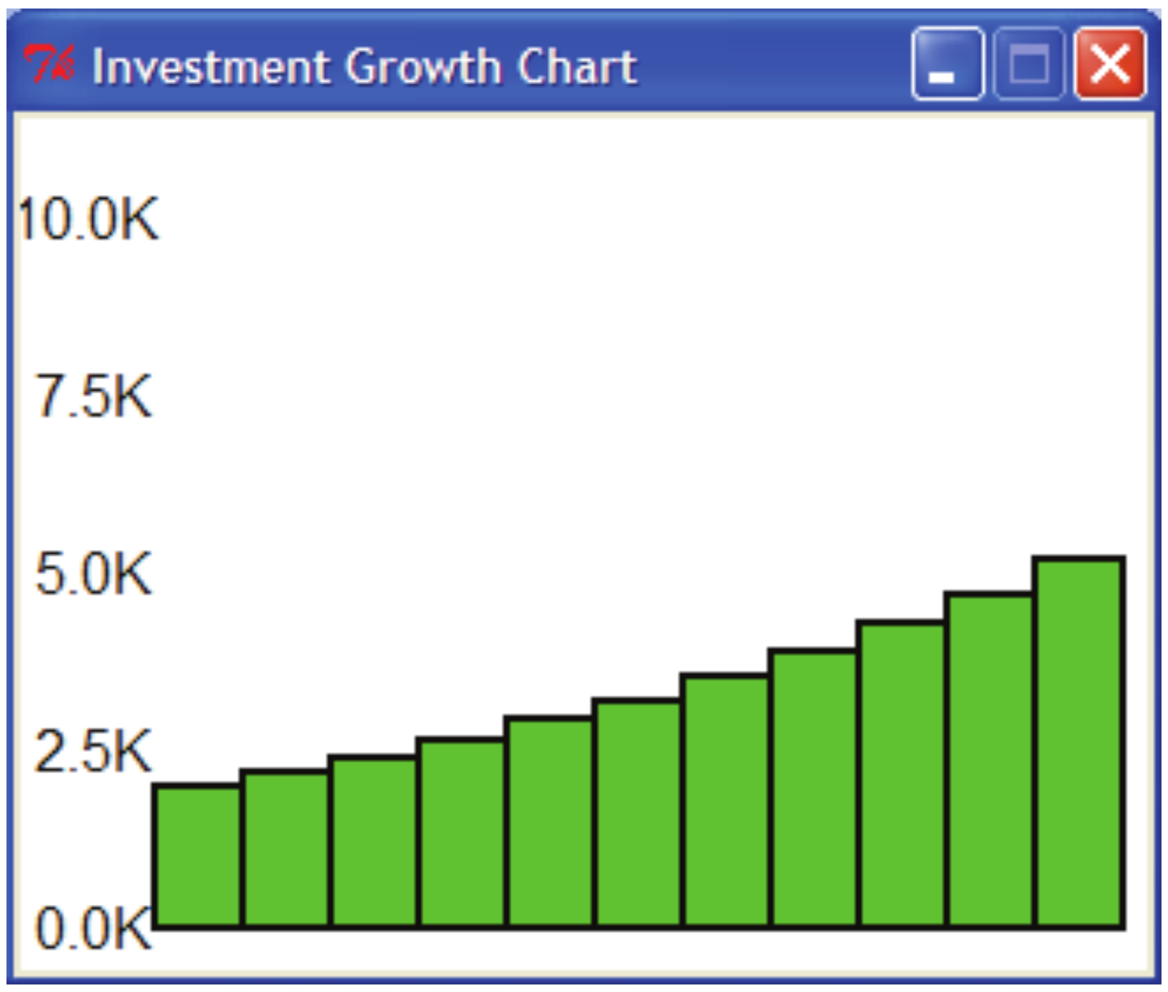


Figure 4.7: Bar graph showing growth of $2000 at 10% interest

Given this analysis, we can flesh out a bit of our design. The third line of the design should now read:

```
Create a 320x240 GraphWin titled ‘‘Investment Growth Chart’’
```

You may be wondering how this will translate into Python code. You have already seen that the `GraphWin` constructor allows an optional parameter to specify the title of the window. You can also supply width and height parameters to control the size of the window. Thus, the command to create the output window will be:

```
win = GraphWin("Investment Growth Chart", 320, 240)
```

Next we turn to the problem of printing labels along the left edge of our window. To simplify the problem, we will assume the graph is always scaled to a maximum of $10,000 with the five labels “0.0K” to “10.0K” as shown in the example window. The question is how should the labels be drawn? We will need some `Text` objects. When creating `Text`, we specify the anchor point (the point the text is centered on) and the string to use as the label.

The label strings are easy. Our longest label is five characters, and the labels should all line up on the right side of a column, so the shorter strings will be padded on the left with spaces. The placement of the labels is chosen with a bit of calculation and some trial and error. Playing with some interactive examples, it seems that a string of length five looks nicely positioned in the horizontal direction placing the center 20 pixels in from the left edge. This leaves just a bit of white space at the margin.

In the vertical direction, we have just over 200 pixels to work with. A simple scaling would be to have 100 pixels represent $5,000. That means our five labels should be spaced 50 pixels apart. Using 200 pixels for the range 0–10,000 leaves $240 - 200 = 40$ pixels to split between the top and bottom margins. We might want to leave a little more margin at the top to accommodate values that grow beyond $10,000. A little experimentation suggests that putting the "0.0K" label 10 pixels from the bottom (position 230) seems to look nice.

Elaborating our algorithm to include these details, the single step

```
Draw scale labels on left side of window
```

becomes a sequence of steps:

```
Draw label " 0.0K" at (20, 230)
Draw label " 2.5K" at (20, 180)
Draw label " 5.0K" at (20, 130)
Draw label " 7.5K" at (20, 80)
Draw label "10.0K" at (20, 30)
```

The next step in the original design calls for drawing the bar that corresponds to the initial amount of the principal. It is easy to see where the lower-left corner of this bar should be. The value of $0.0 is located vertically at pixel 230, and the labels are centered 20 pixels in from the left edge. Adding another 20 pixels gets us to the right edge of the labels. Thus the lower-left corner of the 0th bar should be at location $(40, 230)$.

Now we just need to figure out where the opposite (upper-right) corner of the bar should be so that we can draw an appropriate rectangle. In the vertical direction, the height of the bar is determined by the value of `principal`. In drawing the scale, we determined that 100 pixels is equal to $5,000. This means that we have $100/5000 = 0.02$ pixels to the dollar. This tells us, for example, that a principal of $2,000 should produce a bar of height $2000(.02) = 40$ pixels. In general, the y position of the upper-right corner will be given by $230 - (\text{principal})(0.02)$. (Remember that 230 is the 0 point, and the y coordinates decrease going up.)

How wide should the bar be? The window is 320 pixels wide, but 40 pixels are eaten up by the labels on the left. That leaves us with 280 pixels for 11 bars:

280/11 = 25.4545. Let's just make each bar 25 pixels; that will give us a bit of margin on the right side. So the right edge of our first bar will be at position $40 + 25 = 65$.

We can now fill in the details for drawing the first bar into our algorithm:

```
Draw a rectangle from (40, 230) to (65, 230 - principal * 0.02)
```

At this point, we have made all the major decisions and calculations required to finish out the problem. All that remains is to percolate these details into the rest of the algorithm. Figure 4.8 shows the general layout of the window with some of the dimensions we have chosen.

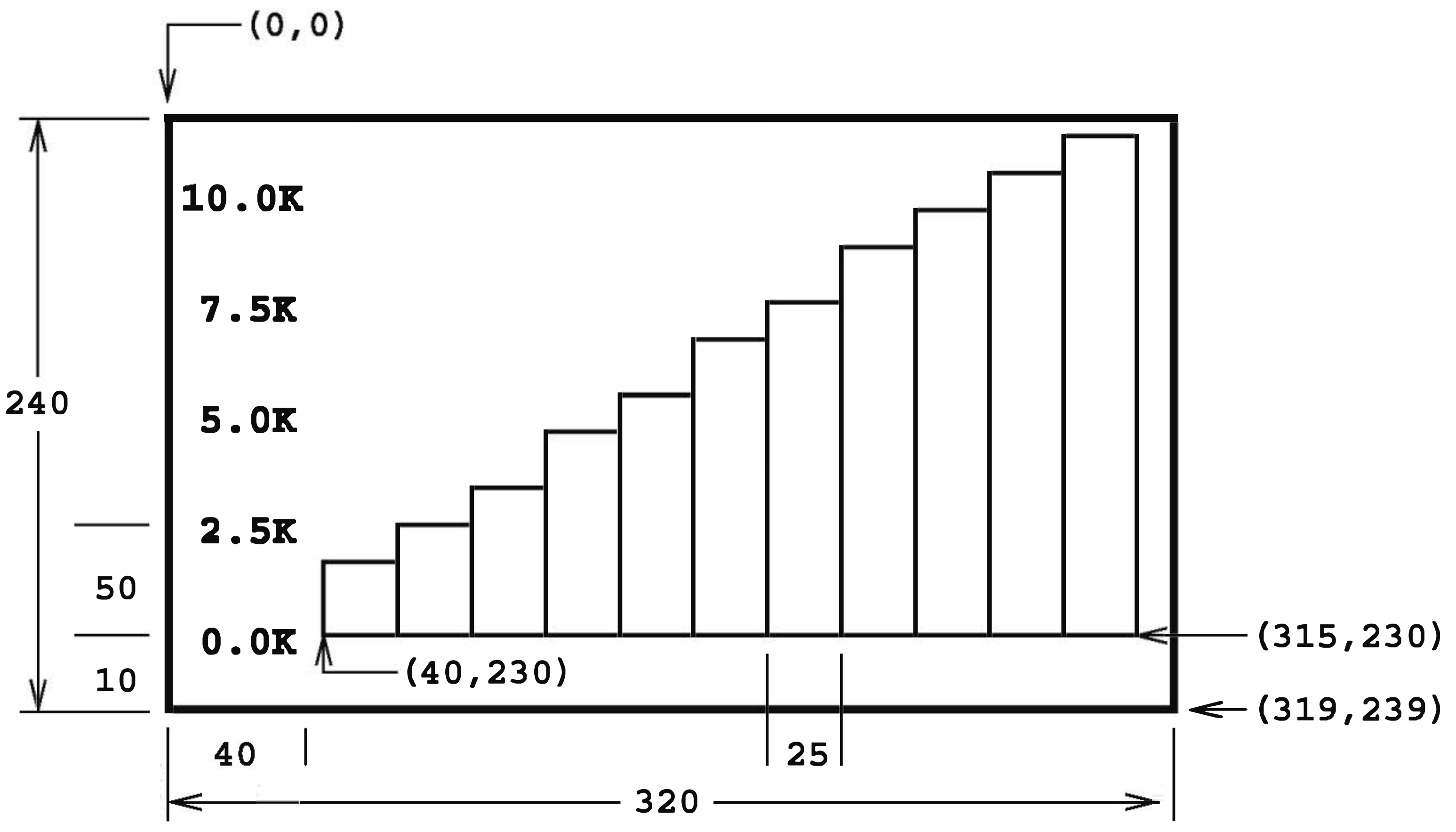


Figure 4.8: Position of elements in future value bar graph

Let's figure out where the lower-left corner of each bar is going to be located. We chose a bar width of 25, so the bar for each successive year will start 25 pixels farther right than the previous year. We can use a variable `year` to represent the year number and calculate the x coordinate of the lower-left corner as (year)(25) + 40. (The +40 leaves space on the left edge for the labels.) Of course, the y coordinate of this point is still 230 (the bottom of the graph).

To find the upper-right corner of a bar, we add 25 (the width of the bar) to the x value of the lower-left corner. The y value of the upper-right corner is determined from the (updated) value of `principal` exactly as we determined it for the first bar. Here is the refined algorithm:

```
for year running from a value of 1 up through 10:
    Calculate principal = principal * (1 + apr)
    Calculate xll = 25 * year + 40
    Calculate height = principal * 0.02
    Draw a rectangle from (xll, 230) to (xll+25, 230 - height)
```

The variable `xll` stands for x lower-left—the x value of the lower-left corner of the bar.

Putting all of this together produces the detailed algorithm shown below:

```
Print an introduction
Get value of principal and apr from user
Create a 320x240 GraphWin titled ‘‘Investment Growth Chart’’
Draw label " 0.0K" at (20, 230)
Draw label " 2.5K" at (20, 180)
Draw label " 5.0K" at (20, 130)
Draw label " 7.5K" at (20, 80)
Draw label "10.0K" at (20, 30)
Draw a rectangle from (40, 230) to (65, 230 - principal * 0.02)
for year running from a value of 1 up through 10:
    Calculate principal = principal * (1 + apr)
    Calculate xll = 25 * year + 40
    Draw a rectangle from (xll, 230) to (xll+25, 230 - principal * 0.02)
Wait for user to press Enter
```

Whew! That was a lot of work, but we are finally ready to translate this algorithm into actual Python code. The translation is straightforward using objects from the graphics library. Here's the program:

```
# futval_graph.py

from graphics import *

def main():
    # Introduction
    print("This program plots the growth of a 10-year investment.")

    # Get principal and interest rate
    principal = float(input("Enter the initial principal: "))
    apr = float(input("Enter the annualized interest rate: "))
    # Create a graphics window with labels on left edge
    win = GraphWin("Investment Growth Chart", 320, 240)
    win.setBackground("white")
    Text(Point(20, 230), ' 0.0K').draw(win)
    Text(Point(20, 180), ' 2.5K').draw(win)
    Text(Point(20, 130), ' 5.0K').draw(win)
    Text(Point(20, 80), ' 7.5K').draw(win)
    Text(Point(20, 30), '10.0K').draw(win)

    # Draw bar for initial principal
    height = principal * 0.02
    bar = Rectangle(Point(40, 230), Point(65, 230-height))
    bar.setFill("green")
    bar.setWidth(2)
    bar.draw(win)

    # Draw bars for successive years
    for year in range(1,11):
        # calculate value for the next year
        principal = principal * (1 + apr)
        # draw bar for this value
        xll = year * 25 + 40
        height = principal * 0.02
        bar = Rectangle(Point(xll, 230), Point(xll+25, 230-height))
        bar.setFill("green")
        bar.setWidth(2)
        bar.draw(win)


    input("Press <Enter> to quit")
    win.close()

main()
```

If you study this program carefully, you will see that I added a number of features to spruce it up a bit. All graphical objects support methods for changing color. I have set the background color of the window to white:

```
win.setBackground("white")
```

I have also changed the color of the `bar` object. The following line asks the bar to color its interior green (because it's money, you know):

```
bar.setFill("green")
```

You can also change the color of a shape's outline using the `setOutline` method. In this case, I have chosen to leave the outline the default black so that the bars stand out from each other. To enhance this effect, this code makes the outline wider (two pixels instead of the default one):

```
bar.setWidth(2)
```

You might also have noted the economy of notation in drawing the labels. Since we don't ever change the labels, assigning them to a variable is unnecessary. We can just create a `Text` object, tell it to draw itself, and be done with it. Here is an example:

```
Text(Point(20,230), ' 0.0K').draw(win)
```

Finally, take a close look at the use of the `year` variable in the loop:

```
for year in range(1,11):
```

The expression `range(1,11)` produces a sequence of ints 1–10. The loop index variable `year` marches through this sequence on successive iterations of the loop. So the first time through `year` is 1, then 2, then 3, etc., up to 10. The value of `year` is then used to compute the proper position of the lower-left corner of each bar:

```
xll = year * 25 + 40
```

I hope you are starting to get the hang of graphics programming. It's a bit strenuous, but very addictive.

4.6 Choosing Coordinates

The lion's share of the work in designing the `futval_graph` program was in determining the precise coordinates where things would be placed on the screen. Most graphics programming problems require some sort of a *coordinate transformation* to change values from a real-world problem into the window coordinates that get mapped onto the computer screen. In our example, the problem domain called for x values representing the year (0–10) and y values representing monetary amounts ($0–$10,000). We had to transform these values to be represented in a 320 x 240 window. It's nice to work through an example or two to see how this transformation happens, but it makes for tedious programming.

Coordinate transformation is an integral and well-studied component of computer graphics. It doesn't take too much mathematical savvy to see that the transformation process always follows the same general pattern. Anything that follows a pattern can be done automatically. In order to save you the trouble of having to explicitly convert back and forth between coordinate systems, the graphics library provides a simple mechanism to do it for you. When you create a `GraphWin` you can specify a coordinate system for the window using the `setCoords` method. The method requires four parameters specifying the coordinates of the lower-left and upper-right corners, respectively. You can then use this coordinate system to place graphical objects in the window.

To take a simple example, suppose we just want to divide the window into nine equal squares, tic-tac-toe fashion. This could be done without too much trouble using the default 200 x 200 window, but it would require a bit of arithmetic. The problem becomes trivial if we first change the coordinates of the window to run from 0 to 3 in both dimensions:

```
# create a default 200x200 window
win = GraphWin("Tic-Tac-Toe")

# set coordinates to go from (0,0) in the lower left
#    to (3,3) in the upper right.
win.setCoords(0.0, 0.0, 3.0, 3.0)

# Draw vertical lines
Line(Point(1,0), Point(1,3)).draw(win)
Line(Point(2,0), Point(2,3)).draw(win)

# Draw horizontal lines
Line(Point(0,1), Point(3,1)).draw(win)
Line(Point(0,2), Point(3,2)).draw(win)
```

Another benefit of this approach is that the size of the window can be changed by simply changing the dimensions used when the window is created (e.g. `win = GraphWin("Tic-Tac-Toe", 300, 300)`). Because the same coordinates span the window (due to `setCoords`) the objects will scale appropriately to the new window size. Using "raw" window coordinates would require changes in the definitions of the lines.

We can apply this idea to simplify our graphing future value program. Basically, we want our graphics window to go from 0 through 10 (representing years) in the x dimension, and from 0 to 10,000 (representing dollars) in the y dimension. We could create just such a window like this:

```
win = GraphWin("Investment Growth Chart", 320, 240)
win.setCoords(0.0, 0.0, 10.0, 10000.0)
```

Then creating a bar for any values of `year` and `principal` would be simple. Each bar starts at the given year and a baseline of 0, and grows to the next year and a height equal to `principal`.

```
bar = Rectangle(Point(year, 0), Point(year+1, principal))
```

There is a small problem with this scheme. Can you see what I have forgotten? The eleven bars will fill the entire window; we haven't left any room for labels or margins around the edges. This is easily fixed by expanding the coordinates of the window slightly. Since our bars start at 0, we can locate the left side labels at -1. We can add a bit of white space around the graph by expanding the coordinates slightly beyond those required for our graph. A little experimentation leads to this window definition:

```
win = GraphWin("Investment Growth Chart", 320, 240)
win.setCoords(-1.75,-200, 11.5, 10400)
```

Here is the program again, using the alternative coordinate system:

```
# futval_graph2.py

from graphics import *

def main():
    # Introduction
    print("This program plots the growth of a 10-year investment.")

    # Get principal and interest rate
    principal = float(input("Enter the initial principal: "))
    apr = float(input("Enter the annualized interest rate: "))
    # Create a graphics window with labels on left edge
    win = GraphWin("Investment Growth Chart", 320, 240)
    win.setBackground("white")
    win.setCoords(-1.75,-200, 11.5, 10400)
    Text(Point(-1, 0), ' 0.0K').draw(win)
    Text(Point(-1, 2500), ' 2.5K').draw(win)
    Text(Point(-1, 5000), ' 5.0K').draw(win)
    Text(Point(-1, 7500), ' 7.5k').draw(win)
    Text(Point(-1, 10000), '10.0K').draw(win)

    # Draw bar for initial principal
    bar = Rectangle(Point(0, 0), Point(1, principal))
    bar.setFill("green")
    bar.setWidth(2)
    bar.draw(win)

    # Draw a bar for each subsequent year
    for year in range(1, 11):
        principal = principal * (1 + apr)
        bar = Rectangle(Point(year, 0), Point(year+1, principal))
        bar.setFill("green")
        bar.setWidth(2)
        bar.draw(win)

    input("Press <Enter> to quit.")
    win.close()

main()
```

Notice how the cumbersome coordinate calculations have been eliminated. This version also makes it easy to change the size of the `GraphWin`. Changing the window size to 640 x 480 produces a larger, but correctly drawn, bar graph. In the original program, all of the calculations would have to be redone to accommodate the new scaling factors in the larger window.

Obviously, the second version of our program is much easier to develop and understand. When you are doing graphics programming, give some consideration to choosing a coordinate system that will make your task as simple as possible.

4.7 Interactive Graphics

Graphical interfaces can be used for input as well as output. In a GUI environment, users typically interact with their applications by clicking on buttons, choosing items from menus, and typing information into on-screen text boxes. These applications use a technique called *event-driven* programming. Basically, the program draws a set of interface elements (often called *widgets*) on the screen, and then waits for the user to do something.

When the user moves the mouse, clicks a button, or types a key on the keyboard, this generates an *event*. Basically, an event is an object that encapsulates data about what just happened. The event object is then sent off to an appropriate part of the program to be processed. For example, a click on a button might produce a *button event*. This event would be passed to the button-handling code, which would then perform the appropriate action corresponding to that button.

Event-driven programming can be tricky for novice programmers, since it's hard to figure out "who's in charge" at any given moment. The graphics module hides the underlying event-handling mechanisms and provides a few simple ways of getting user input in a GraphWin.

4.7.1 Getting Mouse Clicks

We can get graphical information from the user via the getMouse method of the GraphWin class. When getMouse is invoked on a GraphWin, the program pauses and waits for the user to click the mouse somewhere in the graphics window. The spot where the user clicks is returned to the program as a Point. Here is a bit of code that reports the coordinates of ten successive mouse clicks:

```
# click.py
from graphics import *

def main():
    win = GraphWin("Click Me!")
    for i in range(10):
        p = win.getMouse()
        print("You clicked at:", p.getX(), p.getY())

main()
```

The value returned by getMouse() is a ready-made Point. We can use it like any other point using accessors such as `getX` and `getY` or other methods such as `draw` and `move`.

Here is an example of an interactive program that allows the user to draw a triangle by clicking on three points in a graphics window. This example is completely graphical, making use of `Text` objects as prompts. No interaction with a Python text window is required. If you are programming in a Microsoft Windows environment, you can name this program using a `.pyw` extension. Then when the program is run, it will not even display the Python shell window.

```
# triangle.pyw
from graphics import *

def main():
    win = GraphWin("Draw a Triangle")
    win.setCoords(0.0, 0.0, 10.0, 10.0)
    message = Text(Point(5, 0.5), "Click on three points")
    message.draw(win)

    # Get and draw three vertices of triangle
    p1 = win.getMouse()
    p1.draw(win)
    p2 = win.getMouse()
    p2.draw(win)
    p3 = win.getMouse()
    p3.draw(win)

    # Use Polygon object to draw the triangle
    triangle = Polygon(p1,p2,p3)
    triangle.setFill("peachpuff")
    triangle.setOutline("cyan")
    triangle.draw(win)

    # Wait for another click to exit
    message.setText("Click anywhere to quit.")
    win.getMouse()

main()
```

The three-click triangle illustrates a couple of new features of the graphics module. There is no triangle class; however, there is a general class `Polygon` that can be used for any multi-sided, closed shape. The constructor for `Polygon` accepts any number of points and creates a polygon by using line segments to connect the points in the order given and to connect the last point back to the first. A triangle is just a three-sided polygon. Once we have three `Points`—`p1`, `p2`, and `p3`—creating the triangle is a snap:

```
triangle = Polygon(p1, p2, p3)
```

You should also study how the `Text` object is used to provide prompts. A single `Text` object is created and drawn near the beginning of the program:

```
message = Text(Point(5, 0.5), "Click on three points")
message.draw(win)
```

To change the prompt, we don't need to create a new `Text` object; we can just change the text that is displayed. This is done near the end of the program with the `setText` method:

```
message.setText("Click anywhere to quit.")
```

As you can see, the `getMouse` method of `GraphWin` provides a simple way of interacting with the user in a graphics-oriented program.

4.7.2 Handling Textual Input

In the triangle example, all of the input was provided through mouse clicks. Often we will want to allow the user to interact with a graphics window via the keyboard. The `GraphWin` object provides a `getKey()` method that works very much like the `getMouse` method. Here's an extension of the simple clicking program that allows the user to label positions in a window by typing a single keypress after each mouse click:

```
# clickntype.py

from graphics import *

def main():
    win = GraphWin("Click and Type", 400, 400)
    for i in range(10):
        pt = win.getMouse()
        key = win.getKey()
        label = Text(pt, key)
        label.draw(win)

main()
```

Notice what happens in the loop body. First it waits for a mouse click, and the resulting `Point` is saved as the variable `p`. Then the program waits for the user to type a key on the keyboard. The key that is pressed is returned as a string and saved as the variable `key`. For example, if the user presses *g* on the keyboard, then `key` will be the string `'g'`. The `Point` and string are then used to create a text object (called `label`) that is drawn into the window.

You should try out this pogram to get a feel for what the `getKey` method does. In particular, see what strings are returned when you type some of the weirder keys such as <Shift>, <Ctrl>, or the cursor movement keys.

While the `getKey` method is certainly useful, it is not a very practical way of getting an arbitrary string of characters from the user (for example a number or a name). Fortunately, the graphics library provides an `Entry` object that allows the user to actually type input right into a `GraphWin`.

An `Entry` object draws a box on the screen that can contain text. It understands `setText` and `getText` methods just like the `Text` object does. The difference is that the contents of an `Entry` can be edited by the user. Here's a version of the temperature conversion program from Chapter 2 with a graphical user interface:

```
# convert_gui.pyw
# Program to convert Celsius to Fahrenheit using a simple
#   graphical interface.

from graphics import *

def main():
    win = GraphWin("Celsius Converter", 400, 300)
    win.setCoords(0.0, 0.0, 3.0, 4.0)

    # Draw the interface
    Text(Point(1,3), "   Celsius Temperature:").draw(win)
    Text(Point(1,1), "Fahrenheit Temperature:").draw(win)
    inputText = Entry(Point(2.25, 3), 5)
    inputText.setText("0.0")
    inputText.draw(win)
    outputText = Text(Point(2.25,1),"")
    outputText.draw(win)
    button = Text(Point(1.5,2.0),"Convert It")
    button.draw(win)
    Rectangle(Point(1,1.5), Point(2,2.5)).draw(win)

    # wait for a mouse click
    win.getMouse()

    # convert input
    celsius = float(inputText.getText())
    fahrenheit = 9.0/5.0 * celsius + 32

    # display output and change button
    outputText.setText(round(fahrenheit,2))
    button.setText("Quit")

    # wait for click and then quit
    win.getMouse()
    win.close()

main()
```

When run, this produces a window with an entry box for typing in a Celsius temperature and a "button" for doing the conversion. The button is just for show. The program actually just pauses for a mouse click anywhere in the window. Figure 4.9 shows how the window looks when the program starts.

Initially, the `input` entry box is set to contain the value 0.0. The user can delete this value and type in another temperature. The program pauses until the user clicks the mouse. Notice that the point where the user clicks is not even saved; the `getMouse` method is just used to pause the program until the user has a chance to enter a value in the input box.

The program then processes the input in four steps. First, the text in the input box is converted into a number (via `float`). This number is then converted to degrees Fahrenheit. Finally, the resulting number is displayed in the `output` text area. Although `fahrenheit` is a float value, the `setText` method automatically converts it to a string so that it can be displayed in the output text box.

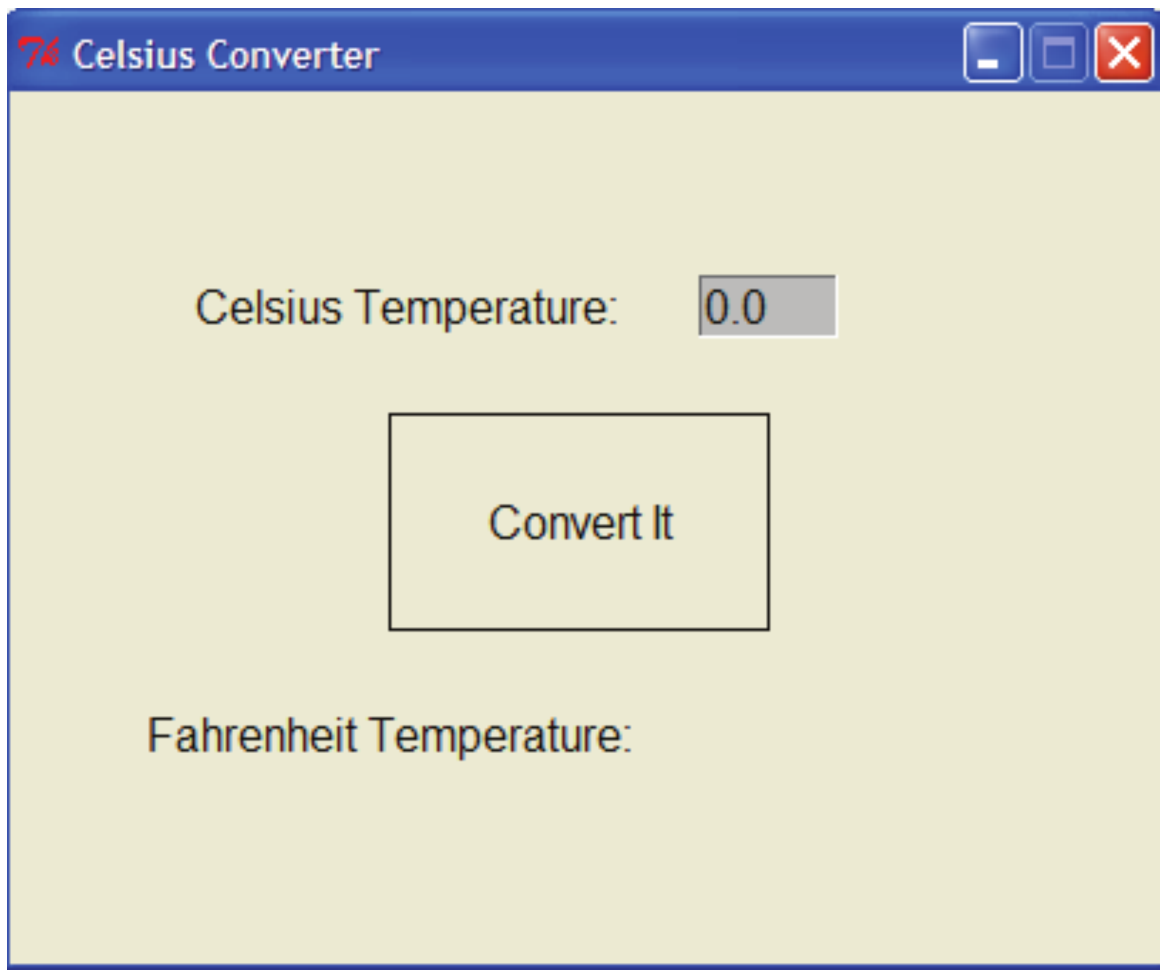


Figure 4.9: Initial screen for graphical temperature converter

Figure 4.10 shows how the window looks after the user has typed an input and clicked the mouse. Notice that the converted temperature shows up in the output area, and the label on the button has changed to "Quit" to show that clicking again will exit the program. This example could be made much prettier using some of the options in the graphics library for changing the colors, sizes, and line widths of the various widgets. The code for the program is deliberately spartan to illustrate just the essential elements of GUI design.

Although the basic tools `getMouse`, `getKey`, and `Entry` do not provide a full-fledged GUI environment, we will see in later chapters how these simple mechanisms can support surprisingly rich interactions.

4.8 Graphics Module Reference

The examples in this chapter have touched on most of the elements in the graphics module. This section provides a complete reference to the objects and functions provided in `graphics`. The set of objects and functions that are provided by a module is sometimes called an *Applications Programming Interface*, or *API*. Experienced programmers study APIs to learn about new libraries. You should probably read this section over once to see what the graphics library has to offer.

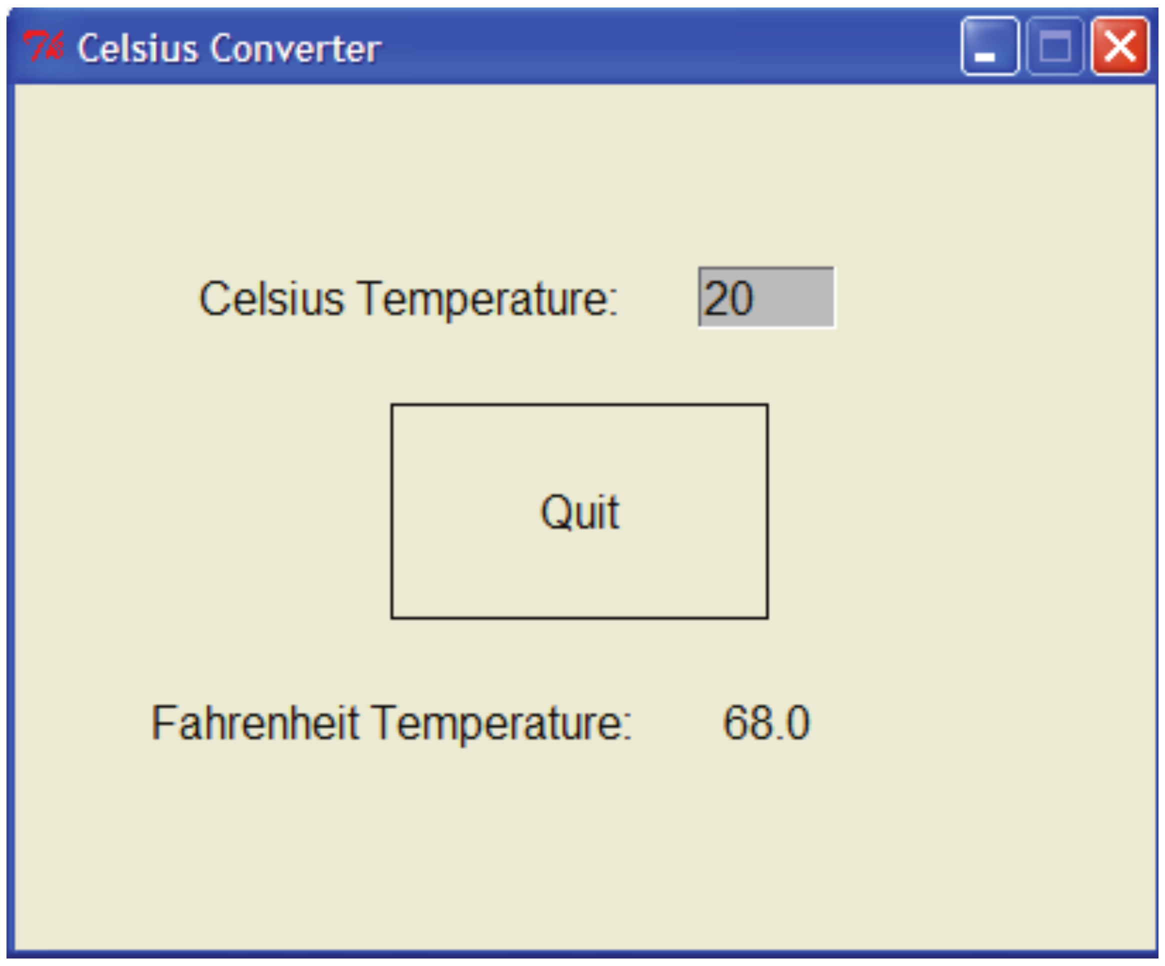


Figure 4.10: Graphical temperature converter after user input

After that, you will probably want to refer back to this section often when you are writing your own graphical programs.

One of the biggest hurdles in learning an API is familiarizing yourself with the various data types that are used. As you read through the reference, pay close attention to the types of the parameters and return values of the various methods. For example, when creating a `Circle`, it's essential that the first parameter you supply *must* be a `Point` object (for the center) and the second parameter *must* be a number (the radius). Using incorrect types will sometimes give an immediate error message, but other times problems may not crop up until later, say when an object is drawn. The examples at the end of each method description incorporate Python literals to illustrate the appropriate data types for parameters.

4.8.1 GraphWin Objects

A `GraphWin` object represents a window on the screen where graphical images may be drawn. A program may define any number of `GraphWin`s. A `GraphWin` understands the following methods:

`GraphWin(title, width, height)` Constructs a new graphics window for drawing on the screen. The parameters are optional; the default title is "Graphics Window," and the default size is 200 x 200 pixels.

Example: `win = GraphWin("Investment Growth", 640, 480)`

`plot(x, y, color)` Draws the pixel at (x, y) in the window. Color is optional; black is the default.

Example: `win.plot(35, 128, "blue")`

`plotPixel(x, y, color)` Draws the pixel at the "raw" position (x, y), ignoring any coordinate transformations set up by `setCoords`.

Example: `win.plotPixel(35, 128, "blue")`

`setBackground(color)` Sets the window background to the given color. The default background color depends on your system. See Section 4.8.5 for information on specifying colors.

Example: `win.setBackground("white")`

`close()` Closes the on-screen window.

Example: `win.close()`

`getMouse()` Pauses for the user to click a mouse in the window and returns where the mouse was clicked as a `Point` object.

Example: `clickPoint = win.getMouse()`

`checkMouse()` Similar to `getMouse`, but does not pause for a user click. Returns the last point where the mouse was clicked or `None`[2] if the window has not been clicked since the previous call to `checkMouse` or `getMouse`. This is particularly useful for controlling animation loops (see Chapter 8).

Example: `clickPoint = win.checkMouse()`
Note: `clickPoint` may be `None`.

`getKey()` Pauses for the user to type a key on the keyboard and returns a string representing the key that was pressed.

Example: `keyString = win.getKey()`

`checkKey()` Similar to `getKey`, but does not pause for the user to press a key. Returns the last key that was pressed or `""` if no key was pressed since the previous call to `checkKey` or `getKey`. This is particularly useful for

[2]`None` is a special Python object often used to signify that a variable has no value. It is discussed in Chapter 6.

controlling simple animation loops (see Chapter 8).

Example: `keyString = win.checkKey()`
Note: `keyString` may be the empty string `""`

`setCoords(xll, yll, xur, yur)` Sets the coordinate system of the window. The lower-left corner is (xll, yll) and the upper-right corner is (xur, yur). Currently drawn objects are redrawn and subsequent drawing is done with respect to the new coordinate system (except for `plotPixel`).

Example: `win.setCoords(0, 0, 200, 100)`

4.8.2 Graphics Objects

The module provides the following classes of drawable objects: `Point`, `Line`, `Circle`, `Oval`, `Rectangle`, `Polygon`, and `Text`. All objects are initially created unfilled with a black outline. All graphics objects support the following generic set of methods:

`setFill(color)` Sets the interior of the object to the given color.

Example: `someObject.setFill("red")`

`setOutline(color)` Sets the outline of the object to the given color.

Example: `someObject.setOutline("yellow")`

`setWidth(pixels)` Sets the width of the outline of the object to the desired number of pixels. (Does not work for `Point`.)

Example: `someObject.setWidth(3)`

`draw(aGraphWin)` Draws the object into the given `GraphWin` and returns the drawn object.

Example: `someObject.draw(someGraphWin)`

`undraw()` Undraws the object from a graphics window. If the object is not currently drawn, no action is taken.

Example: `someObject.undraw()`

`move(dx,dy)` Moves the object `dx` units in the x direction and `dy` units in the y direction. If the object is currently drawn, the image is adjusted to the new position.

Example: `someObject.move(10, 15.5)`

`clone()` Returns a duplicate of the object. Clones are always created in an undrawn state. Other than that, they are identical to the cloned object.

Example: `objectCopy = someObject.clone()`

Point Methods

`Point(x,y)` Constructs a point having the given coordinates.

Example: `aPoint = Point(3.5, 8)`

`getX()` Returns the x coordinate of a point.

Example: `xValue = aPoint.getX()`

`getY()` Returns the y coordinate of a point.

Example: `yValue = aPoint.getY()`

Line Methods

`Line(point1, point2)` Constructs a line segment from `point1` to `point2`.

Example: `aLine = Line(Point(1,3), Point(7,4))`

`setArrow(endString)` Sets the arrowhead status of a line. Arrows may be drawn at either the first point, the last point, or both. Possible values of `endString` are `"first"`, `"last"`, `"both"`, and `"none"`. The default setting is `"none"`.

Example: `aLine.setArrow("both")`

`getCenter()` Returns a clone of the midpoint of the line segment.

Example: `midPoint = aLine.getCenter()`

`getP1(), getP2()` Returns a clone of the corresponding endpoint of the segment.

Example: `startPoint = aLine.getP1()`

Circle Methods

`Circle(centerPoint, radius)` Constructs a circle with the given center point and radius.

Example: `aCircle = Circle(Point(3,4), 10.5)` `getCenter()` Returns a clone of the center point of the circle.

Example: `centerPoint = aCircle.getCenter()`

`getRadius()` Returns the radius of the circle.

Example: `radius = aCircle.getRadius()`

`getP1(), getP2()` Returns a clone of the corresponding corner of the circle's bounding box. These are opposite corner points of a square that circumscribes the circle.

Example: `cornerPoint = aCircle.getP1()`

Rectangle Methods

`Rectangle(point1, point2)` Constructs a rectangle having opposite corners at `point1` and `point2`.

Example: `aRectangle = Rectangle(Point(1,3), Point(4,7))`

`getCenter()` Returns a clone of the center point of the rectangle.

Example: `centerPoint = aRectangle.getCenter()`

`getP1(), getP2()` Returns a clone of the corresponding point used to construct the rectangle.

Example: `cornerPoint = aRectangle.getP1()`

Oval Methods

`Oval(point1, point2)` Constructs an oval in the bounding box determined by `point1` and `point2`.

Example: `anOval = Oval(Point(1,2), Point(3,4))`

`getCenter()` Returns a clone of the point at the center of the oval.

Example: `centerPoint = anOval.getCenter()`

`getP1(), getP2()` Returns a clone of the corresponding point used to construct the oval.

Example: `cornerPoint = anOval.getP1()`

Polygon Methods

`Polygon(point1, point2, point3, ...)` Constructs a polygon with the given points as vertices. Also accepts a single parameter that is a list of the vertices.

Example: `aPolygon = Polygon(Point(1,2), Point(3,4), Point(5,6))`
Example: `aPolygon = Polygon([Point(1,2), Point(3,4), Point(5,6)])`

`getPoints()` Returns a list containing clones of the points used to construct the polygon.

Example: `pointList = aPolygon.getPoints()`

Text Methods

`Text(anchorPoint, textString)` Constructs a text object that displays `textString` centered at `anchorPoint`. The text is displayed horizontally.

Example: `message = Text(Point(3,4), "Hello!")`

`setText(string)` Sets the text of the object to `string`.

Example: `message.setText("Goodbye!")`

`getText()` Returns the current string.

Example: `msgString = message.getText()`

`getAnchor()` Returns a clone of the anchor point.

Example: `centerPoint = message.getAnchor()`

`setFace(family)` Changes the font face to the given `family`. Possible values are `"helvetica"`, `"courier"`, `"times roman"`, and `"arial"`.

Example: `message.setFace("arial")`

`setSize(point)` Changes the font size to the given `point` size. Sizes from 5 to 36 points are legal.

Example: `message.setSize(18)`

`setStyle(style)` Changes font to the given `style`. Possible values are: `"normal"`, `"bold"`, `"italic"`, and `"bold italic"`.

Example: `message.setStyle("bold")`

`setTextColor(color)` Sets the color of the text to `color`. *Note*: `setFill` has the same effect.

Example: `message.setTextColor("pink")`

4.8.3 Entry Objects

Objects of type `Entry` are displayed as text entry boxes that can be edited by the user of the program. `Entry` objects support the generic graphics methods `move()`, `draw(graphwin)`, `undraw()`, `setFill(color)`, and `clone()`. The `Entry` specific methods are given below.

`Entry(centerPoint, width)` Constructs an `Entry` having the given center point and `width`. The `width` is specified in number of characters of text that can be displayed.

Example: `inputBox = Entry(Point(3,4), 5)`

`getAnchor()` Returns a clone of the point where the entry box is centered.

Example: `centerPoint = inputBox.getAnchor()`

`getText()` Returns the string of text that is currently in the entry box.

Example: `inputStr = inputBox.getText()`

`setText(string)` Sets the text in the entry box to the given string.

Example: `inputBox.setText("32.0")`

`setFace(family)` Changes the font face to the given `family`. Possible values are `"helvetica"`, `"courier"`, `"times roman"`, and `"arial"`.

Example: `inputBox.setFace("courier")`

`setSize(point)` Changes the font size to the given `point` size. Sizes from 5 to 36 points are legal.

Example: `inputBox.setSize(12)`

`setStyle(style)` Changes font to the given `style`. Possible values are: `"normal"`, `"bold"`, `"italic"`, and `"bold italic"`.

Example: `inputBox.setStyle("italic")`

`setTextColor(color)` Sets the color of the text to `color`.

Example: `inputBox.setTextColor("green")`

4.8.4 Displaying Images

The graphics module also provides minimal support for displaying and manipulating images in a `GraphWin`. Most platforms will support at least PPM and GIF images. Display is done with an `Image` object. Images support the generic methods `move(dx,dy)`, `draw(graphwin)`, `undraw()`, and `clone()`. Image-specific methods are given below.

`Image(anchorPoint, filename)` Constructs an image from contents of the given file, centered at the given anchor point. Can also be called with `width` and `height` parameters instead of `filename`. In this case, a blank (transparent) image is created of the given width and height (in pixels).

Example: `flowerImage = Image(Point(100,100), "flower.gif")`
Example: `blankImage = Image(320, 240)`

`getAnchor()` Returns a clone of the point where the image is centered.

Example: `centerPoint = flowerImage.getAnchor()`

`getWidth()` Returns the width of the image.

Example: `widthInPixels = flowerImage.getWidth()`

`getHeight()` Returns the height of the image.

Example: `heightInPixels = flowerImage.getHeight()`

`getPixel(x, y)` Returns a list `[red, green, blue]` of the *RGB values* of the pixel at position `(x,y)`. Each value is a number in the range 0–255 indicating the intensity of the corresponding RGB color. These numbers can be turned into a color string using the `color_rgb` function (see next section).

Note that pixel position is relative to the image itself, not the window where the image may be drawn. The upper-left corner of the image is always pixel `(0,0)`.

Example: `red, green, blue = flowerImage.getPixel(32,18)`

`setPixel(x, y, color)` Sets the pixel at position `(x,y)` to the given color. *Note*: This is a slow operation.

Example: `flowerImage.setPixel(32, 18, "blue")`

`save(filename)` Saves the image to a file. The type of the resulting file (e.g., GIF or PPM) is determined by the extension on the filename.

Example: `flowerImage.save("mypic.ppm")`

4.8.5 Generating Colors

Colors are indicated by strings. Most normal colors such as `"red"`, `"purple"`, `"green"`, `"cyan"`, etc. should be available. Many colors come in various shades, such as `"red1"`, `"red2"`,`"red3"`, `"red4"`, which are increasingly darker shades of red. For a full list, look up X11 color names on the web.

The graphics module also provides a function for mixing your own colors numerically. The function `color_rgb(red, green, blue)` will return a string representing a color that is a mixture of the intensities of red, green and blue specified. These should be ints in the range 0–255. Thus `color_rgb(255, 0, 0)` is a bright red, while `color_rgb(130, 0, 130)` is a medium magenta.

Example: `aCircle.setFill(color_rgb(130, 0, 130))`

4.8.6 Controlling Display Updates (Advanced)

Usually, the visual display of a `GraphWin` is updated whenever any graphics object's visible state is changed in some way. However, under some circumstances, for example when using the graphics library inside some interactive shells, it may be necessary to *force* the window to update in order for changes to be seen. The `update()` function is provided to do this.

`update()` Causes any pending graphics operations to be carried out and the results displayed.

For efficiency reasons, it is sometimes desirable to turn off the automatic updating of a window every time one of the objects changes. For example, in an animation, you might want to change the appearance of multiple objects before showing the next “frame” of the animation. The `GraphWin` constructor includes a special extra parameter called `autoflush` that controls this automatic updating. By default, `autoflush` is on when a window is created. To turn it off, the `autoflush` parameter should be set to `False`, like this:

```
win = GraphWin("My Animation", 400, 400, autoflush=False)
```

Now changes to the objects in `win` will only be shown when the graphics system has some idle time or when the changes are forced by a call to `update()`.

The `update()` method also takes an optional parameter that specifies the maximum rate (per second) at which updates can happen. This is useful for controlling the speed of animations in a hardware-independent fashion. For example, placing the command `update(30)` at the bottom of a loop ensures that the loop will "spin" at most 30 times per second. The `update` command will insert an appropriate pause each time through to maintain a relatively constant rate. Of course, the rate throttling will only work when the body of the loop itself executes in less than 1/30th of a second.

Example: 1000 frames at 30 frames per second

```
win = GraphWin("Update Example", 320, 200, autoflush=False)
for i in range(1000):
    # <drawing commands for ith frame>
    update(30)
```

4.9 Chapter Summary

This chapter introduced computer graphics and object-based programming. Here is a summary of some of the important concepts:

- An object is a computational entity that combines data and operations. Objects know stuff and can do stuff. An object's data is stored in instance variables, and its operations are called methods.

- Every object is an instance of some class. It is the class that determines what methods an object will have. An instance is created by calling a constructor method.

- An object's attributes are accessed via dot notation. Generally computations with objects are performed by calling on an object's methods. Accessor methods return information about the instance variables of an object. Mutator methods change the value(s) of instance variables.

- The graphics module supplied with this book provides a number of classes that are useful for graphics programming. A `GraphWin` is an object that represents a window on the screen for displaying graphics. Various graphical objects such as `Point`, `Line`, `Circle`, `Rectangle`, `Oval`, `Polygon`, and `Text` may be drawn in a `GraphWin`. Users may interact with a `GraphWin` by clicking the mouse or typing into an `Entry` box.

- An important consideration in graphical programming is the choice of an appropriate coordinate system. The graphics library provides a way of automating certain coordinate transformations.

- The situation where two variables refer to the same object is called aliasing. Aliasing can sometimes cause unexpected results. Use of the `clone` method in the graphics library can help prevent these situations.

4.10 Exercises

Review Questions

True/False

1. Using `graphics.py` allows graphics to be drawn in a Python shell window.
2. Traditionally, the upper-left corner of a graphics window has coordinates (0,0).
3. A single point on a graphics screen is called a pixel.
4. A function that creates a new instance of a class is called an accessor.
5. Instance variables are used to store data inside an object.
6. The statement `myShape.move(10,20)` moves `myShape` to the point (10,20).
7. Aliasing occurs when two variables refer to the same object.
8. The `copy` method is provided to make a copy of a graphics object.
9. A graphics window always has the title "Graphics Window."
10. The method in the graphics library used to get a mouse click is `readMouse`.

Multiple Choice

1. A method that returns the value of an object's instance variable is called a(n)
   a) mutator b) function c) constructor d) accessor
2. A method that changes the state of an object is called a(n)
   a) stator b) mutator c) constructor d) changor
3. What graphics class would be best for drawing a square?
   a) `Square` b) `Polygon` c) `Line` d) `Rectangle`

4. What command would set the coordinates of `win` to go from (0,0) in the lower-left corner to (10,10) in the upper-right?
   a) `win.setcoords(Point(0,0), Point(10,10))`
   b) `win.setcoords((0,0), (10,10))`
   c) `win.setcoords(0, 0, 10, 10)`
   d) `win.setcoords(Point(10,10), Point(0,0))`

5. What expression would create a line from (2,3) to (4,5)?
   a) `Line(2, 3, 4, 5)`
   b) `Line((2,3), (4,5))`
   c) `Line(2, 4, 3, 5)`
   d) `Line(Point(2,3), Point(4,5))`

6. What command would be used to draw the graphics object `shape` into the graphics window `win`?
   a) `win.draw(shape)` b) `win.show(shape)`
   c) `shape.draw()` d) `shape.draw(win)`

7. Which of the following computes the horizontal distance between points `p1` and `p2`?
   a) `abs(p1-p2)`
   b) `p2.getX() - p1.getX()`
   c) `abs(p1.getY() - p2.getY())`
   d) `abs(p1.getX() - p2.getX())`

8. What kind of object can be used to get text input in a graphics window?
   a) `Text` b) `Entry` c) `Input` d) `Keyboard`

9. A user interface organized around visual elements and user actions is called a(n)
   a) GUI b) application c) windower d) API

10. What color is `color_rgb(0,255,255)`?
    a) yellow b) cyan c) magenta d) orange

Discussion

1. Pick an example of an interesting real-world object and describe it as a programming object by listing its data (attributes, what it "knows") and its methods (behaviors, what it can "do").

2. Describe in your own words the object produced by each of the following operations from the graphics module. Be as precise as you can. Be sure to mention such things as the size, position, and appearance of the various objects. You may include a sketch if that helps.

   a) `Point(130,130)`

   b)
   ```
   c = Circle(Point(30,40),25)
   c.setFill("blue")
   c.setOutline("red")
   ```

   c)
   ```
   r = Rectangle(Point(20,20), Point(40,40))
   r.setFill(color_rgb(0,255,150))
   r.setWidth(3)
   ```

   d)
   ```
   l = Line(Point(100,100), Point(100,200))
   l.setOutline("red4")
   l.setArrow("first")
   ```

   e) `Oval(Point(50,50), Point(60,100))`

   f)
   ```
   shape = Polygon(Point(5,5), Point(10,10), Point(5,10), Point(10,5))
   shape.setFill("orange")
   ```

   g)
   ```
   t = Text(Point(100,100), "Hello World!")
   t.setFace("courier")
   t.setSize(16)
   t.setStyle("italic")
   ```

3. Describe what happens when the following interactive graphics program runs:

```
from graphics import *

def main():
    win = GraphWin()
    shape = Circle(Point(50,50), 20)
    shape.setOutline("red")
    shape.setFill("red")
    shape.draw(win)
    for i in range(10):
        p = win.getMouse()
        c = shape.getCenter()
        dx = p.getX() - c.getX()
```

```
        dy = p.getY() - c.getY()
        shape.move(dx,dy)
    win.close()
main()
```

Programming Exercises

1. Alter the program from the last discussion question in the following ways:

   (a) Make it draw squares instead of circles.

   (b) Have each successive click draw an additional square on the screen (rather than moving the existing one).

   (c) Print a message on the window "Click again to quit" after the loop, and wait for a final click before closing the window.

2. An archery target consists of a central circle of yellow surrounded by concentric rings of red, blue, black and white. Each ring has the same width, which is the same as the radius of the yellow circle. Write a program that draws such a target. *Hint*: Objects drawn later will appear on top of objects drawn earlier.

3. Write a program that draws some sort of face.

4. Write a program that draws a winter scene with a Christmas tree and a snowman.

5. Write a program that draws 5 dice on the screen depicting a straight (1, 2, 3, 4, 5 or 2, 3, 4, 5, 6).

6. Modify the graphical future value program so that the input (principal and APR) also are done in a graphical fashion using `Entry` objects.

7. Circle Intersection.

   Write a program that computes the intersection of a circle with a horizontal line and displays the information textually and graphically.

   Input: Radius of the circle and the y-intercept of the line.

   Output: Draw a circle centered at $(0,0)$ with the given radius in a window with coordinates running from -10,-10 to 10,10.
   Draw a horizontal line across the window with the given y-intercept.
   Draw the two points of intersection in red.
   Print out the x values of the points of intersection.

   Formula: $x = \pm\sqrt{r^2 - y^2}$

8. Line Segment Information.

   This program allows the user to draw a line segment and then displays some graphical and textual information about the line segment.

   Input: Two mouse clicks for the end points of the line segment.

   Output: Draw the midpoint of the segment in cyan.
   Draw the line.
   Print the length and the slope of the line.

   Formulas:
   $$\begin{aligned} dx &= x_2 - x_1 \\ dy &= y_2 - y_1 \\ slope &= dy/dx \\ length &= \sqrt{dx^2 + dy^2} \end{aligned}$$

9. Rectangle Information.

   This program displays information about a rectangle drawn by the user.

   Input: Two mouse clicks for the opposite corners of a rectangle.

   Output: Draw the rectangle.
   Print the perimeter and area of the rectangle.

   Formulas:
   $$\begin{aligned} area &= (length)(width) \\ perimeter &= 2(length + width) \end{aligned}$$

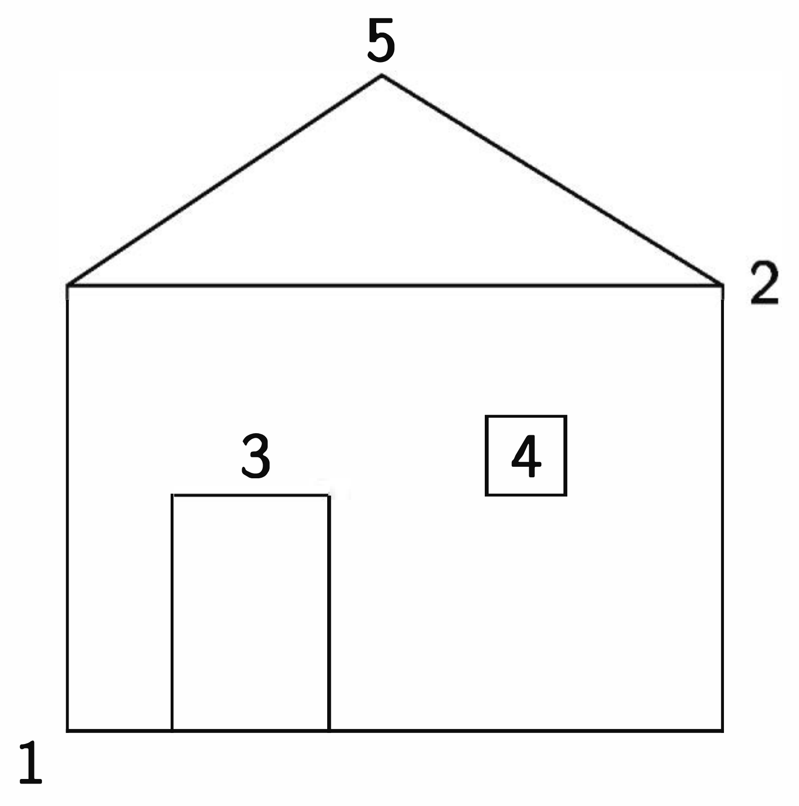
10. Triangle Information.

    Same as the previous problem, but with three clicks for the vertices of a triangle.

    Formulas: For perimeter, see length from the Line Segment problem.
    $area = \sqrt{s(s-a)(s-b)(s-c)}$ where a, b, and c are the lengths of the sides and $s = \frac{a+b+c}{2}$.

11. Five-click House.

    You are to write a program that allows the user to draw a simple house using five mouse clicks. The first two clicks will be the opposite corners of the rectangular frame of the house. The third click will indicate the center of the top edge of a rectangular door. The door should have a total width that is $\frac{1}{5}$ of the width of the house frame. The sides of the door should extend from the corners of the top down to the bottom of the frame. The fourth click will indicate the *center* of a square window. The window is half as wide as the door. The last click will indicate the peak of the roof. The edges of the roof will extend from the point at the peak to the corners of the top edge of the house frame.



Chapter 5 Sequences: Strings, Lists, and Files

Objectives

- To understand the string data type and how strings are represented in the computer.
- To become familiar with various operations that can be performed on strings through built-in functions and string methods.
- To understand the basic idea of sequences and indexing as they apply to Python strings and lists.
- To be able to apply string formatting to produce attractive, informative program output.
- To understand basic file-processing concepts and techniques for reading and writing text files in Python.
- To understand basic concepts of cryptography.
- To understand and write programs that process textual information.

5.1 The String Data Type

So far, we have been discussing programs designed to manipulate numbers and graphics. But you know that computers are also important for storing and operating on textual information. In fact, one of the most common uses for personal computers is word processing. This chapter focuses on textual applications to introduce some important ideas about how text is stored on the computer. You may not think that word-based applications are all that exciting, but as you'll soon see, the basic ideas presented here are at work in virtually all areas of computing, including powering the the World Wide Web.

Text is represented in programs by the *string* data type. You can think of a string as a sequence of characters. In Chapter 2 you learned that a string literal is formed by enclosing some characters in quotation marks. Python also allows strings to be delimited by single quotes (apostrophes). There is no difference; just be sure to use a matching set. Strings can also be saved in variables, just like any other data. Here are some examples illustrating the two forms of string literals:

```
>>> str1 = "Hello"
>>> str2 = 'spam'
>>> print(str1, str2)
Hello spam
>>> type(str1)
<class 'str'>
>>> type(str2)
<class 'str'>
```

You already know how to print strings. You have also seen how to get string input from users. Recall that the `input` function returns whatever the user types as a string object. That means when you want to get a string, you can use the `input` in its "raw" (unconverted) form. Here's a simple interaction to illustrate the point:

```
>>> firstName = input("Please enter your name: ")
Please enter your name: John
>>> print("Hello", firstName)
Hello John
```

Notice how we saved the user's name with a variable and then used that variable to print the name back out again.

So far, we have seen how to get strings as input, assign them to variables, and how to print them out. That's enough to write a parrot program, but not to do any serious text-based computing. For that, we need some string operations. The rest of this section takes you on a tour of the more important Python string operations. In the following section, we'll put these ideas to work in some example programs.

What kinds of things can we do with strings? For starters, remember what a string is: a sequence of characters. One thing we might want to do is access the individual characters that make up the string. In Python, this can be done through the operation of *indexing*. We can think of the positions in a string as being numbered, starting from the left with 0. Figure 5.1 illustrates with the string `Hello Bob`. Indexing is used in string expressions to access a specific character position in the string. The general form for indexing is `<string>[<expr>]`. The value of the expression determines which character is selected from the string.

H	e	l	l	o		B	o	b
0	1	2	3	4	5	6	7	8

Figure 5.1: Indexing of the string `"Hello Bob"`

Here are some interactive indexing examples:

```
>>> greet = "Hello Bob"
>>> greet[0]
'H'
>>> print(greet[0], greet[2], greet[4])
H l o
>>> x = 8
>>> print(greet[x-2])
B
```

Notice that in a string of n characters, the last character is at position $n - 1$, because the indexes start at 0. This is probably also a good time to remind you about the difference between string objects and the actual printed output. In the interactions above, the Python shell shows us the value of strings by putting them in single quotes; that's Python's way of communicating to us that we are looking at a string object. When we actually `print` the string, Python does not put any quotes around the sequence of characters. We just get the text contained in the string.

By the way, Python also allows indexing from the right end of a string using negative indexes.

```
>>> greet[-1]
'b'
```

```
>>> greet[-3]
'B'
```

This is particularly handy for getting at the last character of a string.

Indexing returns a string containing a single character from a larger string. It is also possible to access a contiguous sequence of characters or *substring* from a string. In Python, this is accomplished through an operation called *slicing*. You can think of slicing as a way of indexing a range of positions in the string. Slicing takes the form `<string>[<start>:<end>]`. Both `start` and `end` should be int-valued expressions. A slice produces the substring starting at the position given by `start` and running up to, *but not including*, position `end`.

Continuing with our interactive example, here are some slices:

```
>>> greet[0:3]
'Hel'
>>> greet[5:9]
' Bob'
>>> greet[:5]
'Hello'
>>> greet[5:]
' Bob'
>>> greet[:]
'Hello Bob'
```

The last three examples show that if either expression is missing, the start and end of the string are the assumed defaults. The final expression actually hands back the entire string.

Indexing and slicing are useful operations for chopping strings into smaller pieces. The string data type also supports operations for putting strings together. Two handy operators are concatenation (+) and repetition (`*`). Concatenation builds a string by "gluing" two strings together. Repetition builds a string by multiple concatenations of a string with itself. Another useful function is `len`, which tells how many characters are in a string. Finally, since strings are sequences of characters, you can iterate through the characters using a Python `for` loop.

Here are some examples of various string operations:

```
>>> "spam" + "eggs"
'spameggs'
>>> "Spam" + "And" + "Eggs"
```

```
'SpamAndEggs'
>>> 3 * "spam"
'spamspamspam'
>>> "spam" * 5
'spamspamspamspamspam'
>>> (3 * "spam") + ("eggs" * 5)
'spamspamspameggseggseggseggseggs'
>>> len("spam")
4
>>> len("SpamAndEggs")
11
>>> for ch in "Spam!":
        print(ch, end=" ")
S p a m !
```

These basic string operations are summarized in Table 5.1.

operator	meaning
+	concatenation
*	repetition
<string>[ ]	indexing
<string>[:]	slicing
len(<string>)	length
for <var> in <string>	iteration through characters

Table 5.1: Python string operations

5.2 Simple String Processing

Now that you have an idea what various string operations can do, we're ready to write some programs. Our first example is a program to compute the usernames for a computer system.

Many computer systems use a username and password combination to authenticate system users. The system administrator must assign a unique username to each user. Often, usernames are derived from the user's actual name. One scheme for generating usernames is to use the user's first initial followed by up to seven letters of the user's last name. Using this method, the username for Zaphod Beeblebrox would be “zbeebleb,” and John Smith would just be “jsmith.”

We want to write a program that reads a person’s name and computes the corresponding username. Our program will follow the basic input, process, output pattern. For brevity, I will skip discussion of the algorithm development and jump right to the code. The outline of the algorithm is included as comments in the final program.

```
# username.py
#    Simple string processing program to generate usernames.

def main():
    print("This program generates computer usernames.\n")

    # get user’s first and last names
    first = input("Please enter your first name (all lowercase): ")
    last = input("Please enter your last name (all lowercase): ")

    # concatenate first initial with 7 chars of the last name.
    uname = first[0] + last[:7]

    # output the username
    print("Your username is:", uname)

main()
```

This program first uses `input` to get strings from the user. Then indexing, slicing, and concatenation are combined to produce the username. Here’s an example run:

```
This program generates computer usernames.

Please enter your first name (all lowercase): zaphod
Please enter your last name (all lowercase): beeblebrox
Your username is: zbeebleb
```

Do you see where the blank line between the introduction and the prompt for the first name comes from? Putting the newline character (`\n`) at the end of the string in the first `print` statement caused the output to skip down an extra line. This is a simple trick for putting some extra white space into the output to make it look a little better.

Here is another problem that we can solve with string operations. Suppose we want to print the abbreviation of the month that corresponds to a given month number. The input to the program is an int that represents a month number (1–12), and the output is the abbreviation for the corresponding month. For example, if the input is 3, then the output should be `Mar`, for March.

At first, it might seem that this program is beyond your current ability. Experienced programmers recognize that this is a decision problem. That is, we have to decide which of 12 different outputs is appropriate, based on the number given by the user. We will not cover decision structures until later; however, we can write the program now by some clever use of string slicing.

The basic idea is to store all the month names in a big string:

```
months = "JanFebMarAprMayJunJulAugSepOctNovDec"
```

We can look up a particular month by slicing out the appropriate substring. The trick is computing where to slice. Since each month is represented by three letters, if we knew where a given month started in the string, we could easily extract the abbreviation:

```
monthAbbrev = months[pos:pos+3]
```

This would get us the substring of length 3 that starts in the position indicated by `pos`.

How do we compute this position? Let's try a few examples and see what we find. Remember that string indexing starts at 0.

month	number	position
Jan	1	0
Feb	2	3
Mar	3	6
Apr	4	9

Of course, the positions all turn out to be multiples of 3. To get the correct multiple, we just subtract 1 from the month number and then multiply by 3. So for 1 we get $(1-1)*3 = 0*3 = 0$, and for 12 we have $(12-1)*3 = 11*3 = 33$.

Now we're ready to code the program. Again, the final result is short and sweet; the comments document the algorithm we've developed.

```
# month.py
#  A program to print the abbreviation of a month, given its number
```

```
def main():
    # months is used as a lookup table
    months = "JanFebMarAprMayJunJulAugSepOctNovDec"

    n = int(input("Enter a month number (1-12): "))

    # compute starting position of month n in months
    pos = (n-1) * 3

    # Grab the appropriate slice from months
    monthAbbrev = months[pos:pos+3]

    # print the result
    print("The month abbreviation is", monthAbbrev + ".")

main()
```

Notice the last line of this program uses string concatenation to put a period at the end of the month abbreviation.

Here is a sample of program output:

```
Enter a month number (1-12): 4
The month abbreviation is Apr.
```

One weakness of the "string as lookup table" approach used in this example is that it will only work when the substrings all have the same length (in this case, three). Suppose we want to write a program that outputs the complete month name for a given number. How could that be accomplished?

5.3 Lists as Sequences

Strictly speaking, the operations in Table 5.1 are not really just string operations. They are operations that apply to sequences. As you know from the discussion in Chapter 2, Python lists are also a kind of sequence. That means we can also index, slice, and concatenate lists, as the following session illustrates:

```
>>> [1,2] + [3,4]
[1, 2, 3, 4]
>>> [1,2]*3
```

```
[1, 2, 1, 2, 1, 2]
>>> grades = ['A','B','C','D','F']
>>> grades[0]
'A'
>>> grades[2:4]
['C', 'D']
>>> len(grades)
5
```

One of the nice things about lists is that they are more general than strings. Strings are always sequences of characters, whereas lists can be sequences of arbitrary objects. You can create a list of numbers or a list of strings. In fact, you can even mix it up and create a list that contains both numbers and strings:

```
myList = [1, "Spam", 4, "U"]
```

In later chapters, we'll put all sorts of things into lists like points, rectangles, dice, buttons, and even students!

Using a list of strings, we can rewrite our month abbreviation program from the previous section and make it even simpler:

```
# month2.py
#  A program to print the month abbreviation, given its number.


def main():

    # months is a list used as a lookup table
    months = ["Jan", "Feb", "Mar", "Apr", "May", "Jun",
              "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"]

    n = int(input("Enter a month number (1-12): "))

    print("The month abbreviation is", months[n-1] + ".")

main()
```

There are a couple of things you should notice about this program. I have created a list of strings called `months` to use as the lookup table. The code that creates the list is split over two lines. Normally a Python statement is written on a single line, but in this case Python knows that the list isn't finished until the closing bracket "]" is encountered. Breaking the statement across two lines like this makes the code more readable.

Lists, just like strings, are indexed starting with 0, so in this list the value `months[0]` is the string `"Jan"`. In general, the nth month is at position `n-1`. Since this computation is straightforward, I didn't even bother to put it in a separate step; the expression `months[n-1]` is used directly in the `print` statement.

Not only is this solution to the abbreviation problem a bit simpler, it is also more flexible. For example, it would be trivial to change the program so that it prints out the entire name of the month. All we need is a new definition of the lookup list.

```
    months = ["January", "February", "March", "April",
              "May", "June", "July", "August",
              "September", "October", "November", "December"]
```

While strings and lists are both sequences, there is an important difference between the two. Lists are *mutable*. That means that the value of an item in a list can be modified with an assignment statement. Strings, on the other hand, cannot be changed "in place." Here is an example interaction that illustrates the difference:

```
>>> myList = [34, 26, 15, 10]
>>> myList[2]
15
>>> myList[2] = 0
>>> myList
[34, 26, 0, 10]
>>> myString = "Hello World"
>>> myString[2]
'l'
>>> myString[2] = 'z'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

The first line creates a list of four numbers. Indexing position 2 returns the value 15 (as usual, indexes start at 0). The next command assigns the value 0 to the item in position 2. After the assignment, evaluating the list shows that the new value has replaced the old. Attempting a similar operation on a string produces an error. Strings are not mutable; lists are.

5.4 String Representation and Message Encoding

5.4.1 String Representation

Hopefully, you are starting to get the hang of computing with textual (string) data. However, we haven't yet discussed how computers actually manipulate strings. In Chapter 3, you saw that numbers are stored in binary notation (sequences of 0s and 1s); the computer CPU contains circuitry to do arithmetic with these representations. Textual information is represented in exactly the same way. Underneath, when the computer is manipulating text, it is really no different from number crunching.

To understand this, you might think in terms of messages and secret codes. Consider the age-old grade school dilemma. You are sitting in class and want to pass a note to a friend across the room. Unfortunately, the note must pass through the hands, and in front of the curious eyes, of many classmates before it reaches its final destination. And, of course, there is always the risk that the note could fall into enemy hands (the teacher's). So you and your friend need to design a scheme for encoding the contents of your message.

One approach is to simply turn the message into a sequence of numbers. You could choose a number to correspond to each letter of the alphabet and use the numbers in place of letters. Without too much imagination, you might use the numbers 1–26 to represent the letters a–z. Instead of the word "sourpuss," you would write "18, 14, 20, 17, 15, 20, 18, 18." To those who don't know the code, this looks like a meaningless string of numbers. For you and your friend, however, it represents a word.

This is how a computer represents strings. Each character is translated into a number, and the entire string is stored as a sequence of (binary) numbers in computer memory. It doesn't really matter what number is used to represent any given character as long as the computer is consistent about the encoding/decoding process. In the early days of computing, different designers and manufacturers used different encodings. You can imagine what a headache this was for people transferring data between different systems.

Consider a situation that would result if, say, PCs and Macintosh computers each used their own encoding. If you type a term paper on a PC and save it as a text file, the characters in your paper are represented as a certain sequence of numbers. Then, if the file was read into your instructor's Macintosh computer, the numbers would be displayed on the screen as *different* characters from the ones you typed. The result would be gibberish!

To avoid this sort of problem, computer systems today use industry standard encodings. One important standard is called *ASCII* (American Standard Code for Information Interchange). ASCII uses the numbers 0 through 127 to represent the characters typically found on an (American) computer keyboard, as well as certain special values known as *control codes* that are used to coordinate the sending and receiving of information. For example, the capital letters A–Z are represented by the values 65–90, and the lowercase versions have codes 97–122.

One problem with the ASCII encoding, as its name implies, is that it is American-centric. It does not have symbols that are needed in many other languages. Extended ASCII encodings have been developed by the International Standards Organization to remedy this situation. Most modern systems are moving to *Unicode*, a *much* larger standard that aims to include the characters of nearly all written languages. Python strings support the Unicode Standard, so you can wrangle characters from just about any language, provided your operating system has appropriate fonts for displaying the characters.

Python provides a couple of built-in functions that allow us to switch back and forth between characters and the numeric values used to represent them in strings. The `ord` function returns the numeric ("ordinal") code of a single-character string, while `chr` goes the other direction. Here are some interactive examples:

```
>>> ord("a")
97
>>> ord("A")
65
>>> chr(97)
'a'
>>> chr(90)
'Z'
```

If you're reading very carefully, you might notice that these results are consistent with the ASCII encoding of characters that I mentioned above. By design, Unicode uses the same codes as ASCII for the 127 characters originally defined there. But Unicode includes many more exotic characters as well. For example, the Greek letter pi is character 960, and the symbol for the Euro is character 8364.

There's one more piece in the puzzle of how to store characters in computer memory. As you know from Chapter 3, the underlying CPU deals with memory in fixed-sized pieces. The smallest addressable piece is typically 8 bits, which is called a *byte* of memory. A single byte can store $2^8 = 256$ different values. That's more than enough to represent every possible ASCII character (in fact, ASCII is only a 7-bit code). But a single byte is nowhere near sufficient for storing all the 100,000+ possible Unicode characters. To get around this problem, the Unicode Standard defines various encoding schemes for packing Unicode characters into sequences of bytes. The most common encoding is called *UTF-8*. UTF-8 is a variable-length encoding scheme that uses a single byte to store characters that are in the ASCII subset, but may need up to four bytes in order to represent some of the more esoteric characters. That means that a string of length 10 characters will end up getting stored in memory as a sequence of between 10 and 40 bytes, depending on the actual characters used in the string. As a rule of thumb for Latin alphabets (the usual Western characters), however, it's pretty safe to estimate that a character requires about one byte of storage on average.

5.4.2 Programming an Encoder

Let's return to the note-passing example. Using the Python `ord` and `chr` functions, we can write some simple programs that automate the process of turning messages into sequences of numbers and back again. The algorithm for encoding the message is simple:

```
get the message to encode
for each character in the message:
    print the letter number of the character
```

Getting the message from the user is easy; an `input` will take care of that for us.

```
message = input("Please enter the message to encode: ")
```

Implementing the loop requires a bit more effort. We need to do something for each character of the message. Recall that a `for` loop iterates over a sequence of objects. Since a string is a kind of sequence, we can just use a `for` loop to run through all the characters of the message:

```
for ch in message:
```

Finally, we need to convert each character to a number. The simplest approach is to use the Unicode number (provided by `ord`) for each character in the message.

Here is the final program for encoding the message:

```
# text2numbers.py
#     A program to convert a textual message into a sequence of
#         numbers, utilizing the underlying Unicode encoding.

def main():
    print("This program converts a textual message into a sequence")
    print("of numbers representing the Unicode encoding of the message.\n")

    # Get the message to encode
    message = input("Please enter the message to encode: ")

    print("\nHere are the Unicode codes:")

    # Loop through the message and print out the Unicode values
    for ch in message:
        print(ord(ch), end=" ")

    print() # blank line before prompt

main()
```

We can use the program to encode important messages like this:

```
This program converts a textual message into a sequence
of numbers representing the Unicode encoding of the message.

Please enter the message to encode: What a Sourpuss!

Here are the Unicode codes:
87 104 97 116 32 97 32 83 111 117 114 112 117 115 115 33
```

One thing to notice about this result is that even the space character has a corresponding Unicode number. It is represented by the value 32.

5.5 String Methods

5.5.1 Programming a Decoder

Now that we have a program to turn a message into a sequence of numbers, it would be nice if our friend on the other end had a similar program to turn the numbers back into a readable message. Let's solve that problem next. Our decoder program will prompt the user for a sequence of Unicode numbers and then print out the text message with the corresponding characters. This program presents us with a couple of challenges; we'll address these as we go along.

The overall outline of the decoder program looks very similar to the encoder program. One change in structure is that the decoding version will collect the characters of the message in a string and print out the entire message at the end of the program. To do this, we need to use an accumulator variable, a pattern we saw in the factorial program from Chapter 3. Here is the decoding algorithm:

```
get the sequence of numbers to decode
message = ""
for each number in the input:
    convert the number to the corresponding Unicode character
    add the character to the end of message
print message
```

Before the loop, the accumulator variable `message` is initialized to be an *empty string*; that is, a string that contains no characters (`""`). Each time through the loop, a number from the input is converted into an appropriate character and appended to the end of the message constructed so far.

The algorithm seems simple enough, but even the first step presents us with a problem. How exactly do we get the sequence of numbers to decode? We don't even know how many numbers there will be. To solve this problem, we are going to rely on some more string manipulation operations.

First, we will read the entire sequence of numbers as a single string using `input`. Then we will split the big string into a sequence of smaller strings, each of which represents one of the numbers. Finally, we can iterate through the list of smaller strings, convert each into a number, and use that number to produce the corresponding Unicode character. Here is the complete algorithm:

```
get the sequence of numbers as a string, inString
split inString into a sequence of smaller strings
message = ""
for each of the smaller strings:
    change the string of digits into the number it represents
    append the Unicode character for that number to message
print message
```

This looks complicated, but Python provides some functions that do just what we need.

You may have noticed all along that I've been talking about string objects. Remember from the last chapter, objects have both data and operations (they "know stuff" and "do stuff.") By virtue of being objects, strings have some built-in methods in addition to the generic sequence operations that we have used so far. We'll use some of those abilities here to solve our decoder problem.

For our decoder, we will make use of the `split` method. This method splits a string into a list of substrings. By default, it will split the string wherever a space occurs. Here's an example:

```
>>> myString = "Hello, string methods!"
>>> myString.split()
['Hello,', 'string', 'methods!']
```

Naturally, the `split` operation is called using the usual dot notation for invoking one of an object's methods. In the result, you can see how `split` has turned the original string `"Hello, string methods!"` into a list of three substrings: `"Hello,"`, `"string"`, and `"methods!"`.

By the way, `split` can be used to split a string at places other than spaces by supplying the character to split on as a parameter. For example, if we have a string of numbers separated by commas, we could split on the commas:

```
>>> "32,24,25,57".split(",")
['32', '24', '25', '57']
```

This is useful for getting multiple inputs from the user without resorting to the use of `eval`. For example, we could get the x and y values of a point in a single input string, turn it into a list using the `split` method, and then index into the resulting list to get the individual component strings as illustrated in the following interaction:

```
>>> coords = input("Enter the point coordinates (x,y): ").split(",")
Enter the point coordinates (x,y): 3.4, 6.25
>>> coords
['3.4', '6.25']
>>> coords[0]
'3.4'
>>coords[1]
'6.25'
```

Of course, we still need to convert those strings into the corresponding numbers. Recall from Chapter 3 that we can use the type conversion functions `int` and `float` to convert strings into the appropriate numeric type. In this case, we will use `float` and combine this all into a couple lines of code:

```
coords = input("Enter the point coordinates (x,y): ").split(",")
x,y = float(coords[0]), float(coords[1])
```

Returning to our decoder, we can use a similar technique. Since our program should accept the same format that was produced by the encoder program, namely a sequence of Unicode numbers with spaces between, the default version of `split` works nicely:

```
>>> "87 104 97 116 32 97 32 83 111 117 114 112 117 115 115 33".split()
['87', '104', '97', '116', '32', '97', '32', '83', '111', '117',
'114', '112', '117', '115', '115', '33']
```

Again, the result is not a list of numbers, but a list of strings. It just so happens these strings contain only digits and *could* be interpreted as numbers. In this case, the strings are int literals, so we'll apply the `int` function to each one in order to convert it to a number.

Using `split` and `int` we can write our decoder program:

```
# numbers2text.py
#     A program to convert a sequence of Unicode numbers into
#         a string of text.

def main():
    print("This program converts a sequence of Unicode numbers into")
    print("the string of text that it represents.\n")

    # Get the message to encode
    inString = input("Please enter the Unicode-encoded message: ")

    # Loop through each substring and build Unicode message
    message = ""
    for numStr in inString.split():
        codeNum = int(numStr)            # convert digits to a number
        message = message + chr(codeNum) # concatentate character to message
    print("\nThe decoded message is:", message)

main()
```

Study this program a bit and you should be able to understand exactly how it accomplishes its task. The heart of the program is the loop:

```
for numStr in inString.split():
    codeNum = int(numStr)
    message = message + chr(codeNum)
```

The `split` method produces a list of (sub)strings, and `numStr` takes on each successive string in the list. I called the loop variable `numStr` to emphasize that its value is a string of digits that represents some number. Each time through the loop, the next substring is converted to a number by `int`ing it. This number is converted to the corresponding Unicode character via `chr` and appended to the end of the accumulator, `message`. When the loop is finished, every number in `inString` has been processed and `message` contains the decoded text.

Here is an example of the program in action:

```
This program converts a sequence of Unicode numbers into
the string of text that it represents.

Please enter the Unicode-encoded message:
83 116 114 105 110 103 115 32 97 114 101 32 70 117 110 33

The decoded message is: Strings are Fun!
```

5.5.2 More String Methods

Now we have a couple of programs that can encode and decode messages as sequences of Unicode values. These programs turned out to be quite simple due to the power of both Python's string data type and its built-in sequence operations and string methods.

Python is a very good language for writing programs that manipulate textual data. Table 5.2 lists some other useful string methods. A good way to learn about these operations is to try them out interactively.

```
>>> s = "hello, I came here for an argument"
>>> s.capitalize()
```

```
'Hello, i came here for an argument'
>>> s.title()
'Hello, I Came Here For An Argument'
>>> s.lower()
'hello, i came here for an argument'
>>> s.upper()
'HELLO, I CAME HERE FOR AN ARGUMENT'
>>> s.replace("I", "you")
'hello, you came here for an argument'
>>> s.center(30)
'hello, I came here for an argument'
>>> s.center(50)
'        hello, I came here for an argument        '
>>> s.count('e')
5
>>> s.find(',')
5
>>> " ".join(["Number", "one,", "the", "Larch"])
'Number one, the Larch'
>>> "spam".join(["Number", "one,", "the", "Larch"])
'Numberspamone,spamthespamLarch'
```

I should mention that many of these methods, like `split`, accept additional parameters to customize their operation. Python also has a number of other standard libraries for text processing that are not covered here. You can consult the online documentation or a Python reference to find out more.

5.6 Lists Have Methods, Too

In the last section we took a look at some of the methods for manipulating string objects. Like strings, lists are also objects and come with their own set of "extra" operations. Since this chapter is primarily concerned with text processing, we'll save the detailed discussion of various list methods for a later chapter. However, I do want to introduce one important list method here, just to whet your appetite.

The `append` method can be used to add an item at the end of a list. This is often used to build a list one item at a time. Here's a fragment of code that creates a list of the squares of the first 100 natural numbers:

function	meaning
`s.capitalize()`	Copy of `s` with only the first character capitalized.
`s.center(width)`	Copy of `s` centered in a field of given `width`.
`s.count(sub)`	Count the number of occurrences of `sub` in `s`.
`s.find(sub)`	Find the first position where `sub` occurs in `s`.
`s.join(list)`	Concatenate `list` into a string, using `s` as separator.
`s.ljust(width)`	Like `center`, but `s` is left-justified.
`s.lower()`	Copy of `s` in all lowercase characters.
`s.lstrip()`	Copy of `s` with leading white space removed.
`s.replace(oldsub,newsub)`	Replace all occurrences of `oldsub` in `s` with `newsub`.
`s.rfind(sub)`	Like `find`, but returns the rightmost position.
`s.rjust(width)`	Like `center`, but `s` is right-justified.
`s.rstrip()`	Copy of `s` with trailing white space removed.
`s.split()`	Split `s` into a list of substrings (see text).
`s.title()`	Copy of `s` with first character of each word capitalized.
`s.upper()`	Copy of `s` with all characters converted to uppercase.

Table 5.2: Some string methods

```
squares = []
for x in range(1,101):
    squares.append(x*x)
```

In this example we start with an empty list (`[]`) and each number from 1 to 100 is squared and appended to the list. When the loop is done, `squares` will be the list: `[1, 4, 9, ..., 10000]`. This is really just the accumulator pattern at work again, this time with our accumulated value being a list.

With the `append` method in hand, we can go back and look at an alternative approach to our little decoder program. As we left it, the program used a string variable as an accumulator for the decoded output message. The statement

```
message = message + chr(codeNum)
```

essentially creates a complete copy of the message so far and tacks one more character on the end. As we build up the message, we keep recopying a longer and longer string, just to add a single new character at the end. In older versions of Python, string concatenation could be a slow operation, and programmers often used other techniques to accumulate a long string.

One way to avoid recopying the message over and over again is to use a list. The message can be accumulated as a list of characters where each new character is appended to the end of the existing list. Remember, lists are mutable, so adding at the end of the list changes the list "in place," without having to copy the existing contents over to a new object.[1] Once we have accumulated all the characters in a list, we can use the `join` operation to concatenate the characters into a string in one fell swoop.

Here's a version of the decoder that uses this approach:

```
# numbers2text2.py
#     A program to convert a sequence of Unicode numbers into
#         a string of text. Efficient version using a list accumulator.

def main():
    print("This program converts a sequence of Unicode numbers into")
    print("the string of text that it represents.\n")

    # Get the message to encode
    inString = input("Please enter the Unicode-encoded message: ")

    # Loop through each substring and build Unicode message
    chars = []
    for numStr in inString.split():
        codeNum = int(numStr)            # convert digits to a number
        chars.append(chr(codeNum))        # accumulate new character

    message = "".join(chars)
    print("\nThe decoded message is:", message)

main()
```

In this code, we collect the characters by appending them to a list called `chars`. The final message is obtained by `join`ing these characters together using an empty string as the separator. So the original characters are concatenated together without any extra spaces between.

Both the string concatenation and the `append`/`join` techniques are quite efficient in modern Python, and the choice between them is largely a matter of

[1]Actually, the list does need to be recopied behind the scenes in the case where Python runs out of room for the new item, but this is a rare occurrence.

taste. The list technique is a bit more flexible in that the `join` method makes it easy to build strings that use a special separator (e.g., a tab, comma, or space), if desired, between the concatenated items.

5.7 From Encoding to Encryption

We have looked at how computers represent strings as a sort of encoding problem. Each character in a string is represented by a number that is stored in the computer as a binary representation. You should realize that there is nothing really secret about this code at all. In fact, we are simply using an industry-standard mapping of characters into numbers. Anyone with a little knowledge of computer science would be able to crack our code with very little effort.

The process of encoding information for the purpose of keeping it secret or transmitting it privately is called *encryption*. The study of encryption methods is an increasingly important sub-field of mathematics and computer science known as *cryptography*. For example, if you shop over the Internet, it is important that your personal information such as your name and credit card number be transmitted using encodings that keep it safe from potential eavesdroppers on the network.

Our simple encoding/decoding programs use a very weak form of encryption known as a *substitution cipher*. Each character of the original message, called the *plaintext*, is replaced by a corresponding symbol (in our case a number) from a *cipher alphabet*. The resulting code is called the *ciphertext*.

Even if our cipher were not based on the well-known Unicode encoding, it would still be easy to discover the original message. Since each letter is always encoded by the same symbol, a codebreaker could use statistical information about the frequency of various letters and some simple trial and error testing to discover the original message. Such simple encryption methods may be sufficient for grade-school note passing, but they are certainly not up to the task of securing communication over global networks.

Modern approaches to encryption start by translating a message into numbers, much like our encoding program. Then sophisticated mathematical algorithms are employed to transform these numbers into other numbers. Usually, the transformation is based on combining the message with some other special value called the *key*. In order to decrypt the message, the party on the receiving end needs to have an appropriate key so that the encoding can be reversed to recover the original message.

Encryption approaches come in two flavors: *private key* and *public key*. In a private key (also called *shared key*) system, the same key is used for encrypting and decrypting messages. All parties that wish to communicate need to know the key, but it must be kept secret from the outside world. This is the usual system that people think of when considering secret codes.

In public key systems, there are separate but related keys for encrypting and decrypting. Knowing the encryption key does not allow you to decrypt messages or discover the decryption key. In a public key system, the encryption key can be made publicly available, while the decryption key is kept private. Anyone can safely send a message using the public key for encryption. Only the party holding the decryption key will be able to decipher it. For example, a secure website can send your web browser its public key, and the browser can use it to encode your credit card information before sending it on the Internet. Then only the company that is requesting the information will be able to decrypt and read it using the proper private key.

5.8 Input/Output as String Manipulation

Even programs that we may not view as primarily doing text manipulation often need to make use of string operations. For example, consider a program that does financial analysis. Some of the information (e.g., dates) must be entered as strings. After doing some number crunching, the results of the analysis will typically be a nicely formatted report including textual information that is used to label and explain numbers, charts, tables, and figures. String operations are needed to handle these basic input and output tasks.

5.8.1 Example Application: Date Conversion

As a concrete example, let's extend our month abbreviation program to do date conversions. The user will input a date such as "05/24/2020," and the program will display the date as "May 24, 2020." Here is the algorithm for our program:

```
Input the date in mm/dd/yyyy format (dateStr)
Split dateStr into month, day and year strings
Convert the month string into a month number
Use the month number to look up the month name
Create a new date string in form Month Day, Year
Output the new date string
```

We can implement the first two lines of our algorithm directly in code using string operations we have already discussed:

```
dateStr = input("Enter a date (mm/dd/yyyy): ")
monthStr, dayStr, yearStr = dateStr.split("/")
```

Here I have gotten the date as a string and split it at the slashes. I then "unpacked" the list of three strings into the variables `monthStr`, `dayStr`, and `yearStr` using simultaneous assignment.

The next step is to convert `monthStr` into an appropriate number (using `int` again) and then use this value to look up the correct month name. Here is the code:

```
months = ["January", "February", "March", "April",
          "May", "June", "July", "August",
          "September", "October", "November", "December"]
monthStr = months[int(monthStr)-1]
```

Remember the indexing expression `int(monthStr)-1` is used because list indexes start at 0.

The last step in our program is to piece together the date in the new format:

```
print("The converted date is:", monthStr, dayStr+",", yearStr)
```

Notice how I have used concatenation for the comma immediately after the day.

Here's the complete program:

```
# dateconvert.py
#    Converts a date in form "mm/dd/yyyy" to "month day, year"

def main():
    # get the date
    dateStr = input("Enter a date (mm/dd/yyyy): ")

    # split into components
    monthStr, dayStr, yearStr = dateStr.split("/")

    # convert monthStr to the month name
    months = ["January", "February", "March", "April",
              "May", "June", "July", "August",
              "September", "October", "November", "December"]
    monthStr = months[int(monthStr)-1]

    # output result in month day, year format
    print("The converted date is:", monthStr, dayStr+",", yearStr)

main()
```

When run, the output looks like this:

```
Enter a date (mm/dd/yyyy): 05/24/2020
The converted date is: May 24, 2020
```

This example didn't show it, but often it is also necessary to turn a number into a string. In Python, most data types can be converted into strings using the `str` function. Here are a couple of simple examples:

```
>>> str(500)
'500'
>>> value = 3.14
>>> str(value)
'3.14'
>>> print("The value is", str(value) + ".")
The value is 3.14.
```

Notice particularly the last example. By turning `value` into a string, we can use string concatenation to put a period at the end of a sentence. If we didn't first turn `value` into a string, Python would interpret the + as a numerical operation and produce an error, because "." is not a number.

We now have a complete set of operations for converting values among various Python data types. Table 5.3 summarizes these four Python type conversion functions:

function	meaning
`float(<expr>)`	Convert `expr` to a floating-point value.
`int(<expr>)`	Convert `expr` to an integer value.
`str(<expr>)`	Return a string representation of `expr`.
`eval(<string>)`	Evaluate `string` as an expression.

Table 5.3: Type conversion functions

One common reason for converting a number into a string is so that string operations can be used to control the way the value is printed. For example, a program performing date calculations would have to manipulate the month, day, and year as numbers. For nicely formatted output, these numbers would be converted back to strings.

5.8.2 String Formatting

As you have seen, basic string operations can be used to build nicely formatted output. This technique is useful for simple formatting, but building up a complex output through slicing and concatenation of smaller strings can be tedious. Python provides a powerful string formatting operation that makes the job much easier.

Let's start with a simple example. Here is a run of the change-counting program from Chapter 3:

```
Change Counter

Please enter the count of each coin type.
How many quarters do you have? 6
How many dimes do you have? 0
How many nickels do you have? 0
How many pennies do you have? 0
The total value of your change is 1.5
```

Notice that the final value is given as a fraction with only one decimal place. This looks funny, since we expect the output to be something like `$1.50`.

We can fix this problem by changing the very last line of the program as follows:

```
print("The total value of your change is ${0:0.2f}".format(total))
```

Now the program prints this message:

```
The total value of your change is $1.50
```

Let's try to make some sense of this. The `format` method is a built-in for Python strings. The idea is that the string serves as a sort of template, and values supplied as parameters are plugged into this template to form a new string. So string formatting takes the form:

```
<template-string>.format(<values>)
```

Curly braces ({}) inside the `template-string` mark "slots" into which the provided `values` are inserted. The information inside the curly braces tells which value goes in the slot and how the value should be formatted. The Python formatting operator is very flexible. We will cover just some basics here; you can consult a Python reference if you'd like all of the details. In this book, the slot descriptions will always have the form:

```
{<index>:<format-specifier>}
```

The `index` tells which of the parameters is inserted into the slot.[2] As usual in Python, indexing starts with 0. In the example above, there is a single slot and the index 0 is used to say that the first (and only) parameter is inserted into that slot.

The part of the description after the colon specifies how the value should look when it is inserted into the slot. Again returning to the example, the format specifier is `0.2f`. The format of this specifier is `<width>.<precision><type>`. The width specifies how many "spaces" the value should take up. If the value takes up less than the specified width, it is padded with extra characters (spaces are the default). If the value requires more space than allotted, it will take as much space as is required to show the value. So putting a 0 here essentially says "use as much space as you need." The precision is 2, which tells Python to round the value to two decimal places. Finally, the type character `f` says the value should be displayed as a fixed-point number. That means that the specified number of decimal places will always be shown, even if they are 0.

A complete description of format specifiers is pretty hairy, but you can get a good handle on what's possible just by looking at a few examples. The simplest template strings just specify where to plug in the parameters.

```
>>> "Hello {0} {1}, you may have won ${2}".format("Mr.", "Smith", 10000)
'Hello Mr. Smith, you may have won $10000'
```

Often, you'll want to control the width and/or precision of a numeric value.

```
>>> "This int, {0:5}, was placed in a field of width 5".format(7)
'This int,     7, was placed in a field of width 5'

>>> "This int, {0:10}, was placed in a field of width 10".format(7)
'This int,          7, was placed in a field of width 10'
```

[2]As of Python 3.1, the index portion of the slot description is optional. When the indexes are omitted, the parameters are just filled into the slots in a left-to-right fashion.

```
>>> "This float, {0:10.5}, has width 10 and precision 5".format(3.1415926)
'This float,     3.1416, has width 10 and precision 5'

>>> "This float, {0:10.5f}, is fixed at 5 decimal places".format(3.1415926)
'This float,    3.14159, is fixed at 5 decimal places'

>>> "This float, {0:0.5}, has width 0 and precision 5".format(3.1415926)
'This float, 3.1416, has width 0 and precision 5'

>>> "Compare {0} and {0:0.20}".format(3.14)
'Compare 3.14 and 3.1400000000000001243'
```

Notice that for normal (not fixed-point) floating-point numbers, the precision specifies the number of significant digits to print. For fixed-point (indicated by the `f` at the end of the specifier) the precision gives the number of decimal places. In the last example, the same number is printed out in two different formats. This illustrates that if you print enough digits of a floating-point number, you will almost always find a "surprise." The computer can't represent 3.14 exactly as a floating-point number. The closest value it can represent is ever so slightly larger than 3.14. If not given an explicit precision, Python will print the number out to a few decimal places. The slight extra amount shows up if you print lots of digits. Generally, Python only displays a closely rounded version of a float. Using explicit formatting allows you to see the full result down to the last bit.

You may notice that, by default, numeric values are right-justified. This is helpful for lining up numbers in columns. Strings, on the other hand, are left-justified in their fields. You can change the default behaviors by including an explicit justification character at the beginning of the format specifier. The necessary characters are <, >, and ^ for left, right, and center justification, respectively.

```
>>> "left justification: {0:<5}".format("Hi!")
'left justification: Hi!  '

>>> "right justification: {0:>5}".format("Hi!")
'right justification:   Hi!'

>>> "centered: {0:^5}".format("Hi!")
'centered:  Hi! '
```

5.8.3 Better Change Counter

Let's close our formatting discussion with one more example program. Given what you have learned about floating-point numbers, you might be a little uneasy about using them to represent money.

Suppose you are writing a computer system for a bank. Your customers would not be too happy to learn that a charge went through for an amount "very close to $107.56." They want to know that the bank is keeping precise track of their money. Even though the amount of error in a given value is very small, the small errors can be compounded when doing lots of calculations, and the resulting error could add up to some real cash. That's not a satisfactory way of doing business.

A better approach would be to make sure that our program uses exact values to represent money. We can do that by keeping track of the money in cents and using an int to store it. We can then convert this into dollars and cents in the output step. Assuming we are dealing with positive amounts, if `total` represents the value in cents, then we can get the number of dollars by integer division `total // 100` and the cents from `total % 100`. Both of these are integer calculations and, hence, will give us exact results. Here is the updated program:

```
# change2.py
#   A program to calculate the value of some change in dollars
#   This version represents the total cash in cents.

def main():
    print("Change Counter\n")
    print("Please enter the count of each coin type.")
    quarters = int(input("Quarters: "))
    dimes = int(input("Dimes: "))
    nickels = int(input("Nickels: "))
    pennies = int(input("Pennies: "))

    total = quarters * 25 + dimes * 10 + nickels * 5 + pennies

    print("The total value of your change is ${0}.{1:0>2}"
          .format(total//100, total%100))

main()
```

I have split the final `print` statement across two lines. Normally a statement ends at the end of the line, but sometimes it is nicer to break a long statement into smaller pieces. Because this line is broken in the middle of the `print` function, Python knows that the statement is not finished until the final closing parenthesis is reached. In this case, it is OK, and preferable, to break the statement across two lines rather than having one really long line.

The string formatting in the print statement contains two slots, one for dollars as an int and one for cents. The cents slot illustrates one additional twist on format specifiers. The value of cents is printed with the specifier `0>2`. The zero in front of the justification character tells Python to pad the field (if necessary) with zeroes instead of spaces. This ensures that a value like 10 dollars and 5 cents prints as `$10.05` rather than `$10. 5`.

5.9 File Processing

I began the chapter with a reference to word processing as an application of the string data type. One critical feature of any word processing program is the ability to store and retrieve documents as files on disk. In this section, we'll take a look at file input and output, which, as it turns out, is really just another form of string processing.

5.9.1 Multi-line Strings

Conceptually, a file is a sequence of data that is stored in secondary memory (usually on a disk drive). Files can contain any data type, but the easiest files to work with are those that contain text. Files of text have the advantage that they can be read and understood by humans, and they are easily created and edited using general-purpose text editors (such as IDLE) and word processors. In Python, text files can be very flexible, since it is easy to convert back and forth between strings and other types.

You can think of a text file as a (possibly long) string that happens to be stored on disk. Of course, a typical file generally contains more than a single line of text. A special character or sequence of characters is used to mark the end of each line. There are numerous conventions for end-of-line markers. Python takes care of these different conventions for us and just uses the regular newline character (`\n`) to indicate line breaks.

Let's take a look at a concrete example. Suppose you type the following lines into a text editor exactly as shown here:

```
Hello
World

Goodbye 32
```

When stored to a file, you get this sequence of characters:

```
Hello\nWorld\n\nGoodbye 32\n
```

Notice that the blank line becomes a bare newline in the resulting file/string.

By the way, this is really no different than when we embed newline characters into output strings to produce multiple lines of output with a single `print` statement. Here is the example from above printed interactively:

```
>>> print("Hello\nWorld\n\nGoodbye 32\n")
Hello
World

Goodbye 32

>>>
```

Remember, if you simply evaluate a string containing newline characters in the shell, you will just get the embedded newline representation back again:

```
>>>"Hello\nWorld\n\nGoodbye 32\n"
'Hello\nWorld\n\nGoodbye 32\n'
```

It's only when a string is printed that the special characters affect how the string is displayed.

5.9.2 File Processing

The exact details of file processing differ substantially among programming languages, but virtually all languages share certain underlying file-manipulation concepts. First, we need some way to associate a file on disk with an object in a program. This process is called *opening* a file. Once a file has been opened, its contents can be accessed through the associated file object.

Second, we need a set of operations that can manipulate the file object. At the very least, this includes operations that allow us to read the information from a file and write new information to a file. Typically, the reading and writing operations for text files are similar to the operations for text-based, interactive input and output.

Finally, when we are finished with a file, it is *closed*. Closing a file makes sure that any bookkeeping that was necessary to maintain the correspondence between the file on disk and the file object is finished up. For example, if you write information to a file object, the changes might not show up on the disk version until the file has been closed.

This idea of opening and closing files is closely related to how you might work with files in an application program like a word processor. However, the concepts are not exactly the same. When you open a file in a program like Microsoft Word, the file is actually read from the disk and stored into RAM. In programming terminology, the file is opened for reading and the contents of the file are then read into memory via file-reading operations. At this point, the file is closed (again in the programming sense). As you "edit the file," you are really making changes to data in memory, not the file itself. The changes will not show up in the file on the disk until you tell the application to "save" it.

Saving a file also involves a multi-step process. First, the original file on the disk is reopened, this time in a mode that allows it to store information—the file on disk is opened for writing. Doing so actually *erases* the old contents of the file. File writing operations are then used to copy the current contents of the in-memory version into the new file on the disk. From your perspective, it appears that you have edited an existing file. From the program's perspective, you have actually opened a file, read its contents into memory, closed the file, created a new file (having the same name), written the (modified) contents of memory into the new file, and closed the new file.

Working with text files is easy in Python. The first step is to create a file object corresponding to a file on disk. This is done using the `open` function. Usually, a file object is immediately assigned to a variable like this:

```
<variable> = open(<name>, <mode>)
```

Here `name` is a string that provides the name of the file on the disk. The `mode` parameter is either the string `"r"` or `"w"` depending on whether we intend to *read* from the file or *write* to the file.

For example, to open a file called "numbers.dat" for reading, we could use a statement like the following:

```
infile = open("numbers.dat", "r")
```

Now we can use the file object `infile` to read the contents of `numbers.dat` from the disk.

Python provides three related operations for reading information from a file:

`<file>.read()` Returns the entire remaining contents of the file as a single (potentially large, multi-line) string.

`<file>.readline()` Returns the next line of the file. That is, all text up to *and including* the next newline character.

`<file>.readlines()` Returns a list of the remaining lines in the file. Each list item is a single line including the newline character at the end.

Here's an example program that prints the contents of a file to the screen using the `read` operation:

```
# printfile.py
#     Prints a file to the screen.

def main():
    fname = input("Enter filename: ")
    infile = open(fname,"r")
    data = infile.read()
    print(data)

main()
```

The program first prompts the user for a file name and then opens the file for reading through the variable `infile`. You could use any name for the variable; I used `infile` to emphasize that the file was being used for input. The entire contents of the file is then read as one large string and stored in the variable `data`. Printing `data` causes the contents to be displayed.

The `readline` operation can be used to read the next line from a file. Successive calls to `readline` get successive lines from the file. This is analogous to `input`, which reads characters interactively until the user hits the `<Enter>` key; each call to `input` gets another line from the user. One thing to keep in mind, however, is that the string returned by `readline` will always end with a newline character, whereas `input` discards the newline character.

As a quick example, this fragment of code prints out the first five lines of a file:

```
infile = open(someFile, "r")
for i in range(5):
    line = infile.readline()
    print(line[:-1])
```

Notice the use of slicing to strip off the newline character at the end of the line. Since `print` automatically jumps to the next line (i.e., it outputs a newline), printing with the explicit newline at the end would put an extra blank line of output between the lines of the file. Alternatively, you could print the whole line, but simply tell `print` not to add its own newline character.

```
print(line, end="")
```

One way to loop through the entire contents of a file is to read in all of the file using `readlines` and then loop through the resulting list:

```
infile = open(someFile, "r")
for line in infile.readlines():
    # process the line here
infile.close()
```

Of course, a potential drawback of this approach is the fact that the file may be very large, and reading it into a list all at once may take up too much RAM.

Fortunately, there is a simple alternative. Python treats the file itself as a sequence of lines. So looping through the lines of a file can be done directly like this:

```
infile = open(someFile, "r")
for line in infile:
    # process the line here
infile.close()
```

This is a particularly handy way to process the lines of a file one at a time.

Opening a file for writing prepares that file to receive data. If no file with the given name exists, a new file will be created. A word of warning: if a file with the given name *does* exist, Python will delete it and create a new, empty file. When writing to a file, make sure you do not clobber any files you will need later! Here is an example of opening a file for output:

```
outfile = open("mydata.out", "w")
```

The easiest way to write information into a text file is to use the already-familiar `print` function. To print to a file, we just need to add an extra keyword parameter that specifies the file:

```
print(..., file=<outputFile>)
```

This behaves exactly like a normal `print` except that the result is sent to `outputFile` instead of being displayed on the screen.

5.9.3 Example Program: Batch Usernames

To see how all these pieces fit together, let's redo the username generation program. Our previous version created usernames interactively by having the user type in his or her name. If we were setting up accounts for a large number of users, the process would probably not be done interactively, but in *batch* mode. In batch processing, program input and output is done through files.

Our new program is designed to process a file of names. Each line of the input file will contain the first and last names of a new user separated by one or more spaces. The program produces an output file containing a line for each generated username:

```
# userfile.py
#    Program to create a file of usernames in batch mode.

def main():
    print("This program creates a file of usernames from a")
    print("file of names.")

    # get the file names
    infileName = input("What file are the names in? ")
    outfileName = input("What file should the usernames go in? ")

    # open the files
    infile = open(infileName, "r")
    outfile = open(outfileName, "w")

    # process each line of the input file
    for line in infile:
        # get the first and last names from line
        first, last = line.split()
        # create the username
        uname = (first[0]+last[:7]).lower()
        # write it to the output file
        print(uname, file=outfile)

    # close both files
    infile.close()
    outfile.close()

    print("Usernames have been written to", outfileName)

main()
```

There are a couple of things worth noticing in this program. I have two files open at the same time, one for input (`infile`) and one for output (`outfile`). It's not unusual for a program to operate on several files simultaneously. Also, when creating the username, I used the `lower` string method. Notice that the method is applied to the string that results from the concatenation. This ensures that the username is all lowercase, even if the input names are mixed case.

5.9.4 File Dialogs (Optional)

One problem that often crops up with file manipulation programs is figuring out exactly how to specify the file that you want to use. If a data file is in the same directory (folder) as your program, then you simply have to type in the correct name of the file; with no other information, Python will look for the file in the "current" directory. Sometimes, however, it's difficult to know exactly what the file's complete name is. Most modern operating systems use file names having a form like `<name>.<type>` where the `type` portion is a short (3- or 4-letter) extension that describes what sort of data the file contains. For example, our usernames might be stored in a file called "users.txt" where the ".txt" extension indicates a text file. The difficulty is that some operating systems (e.g. Windows and macOS), by default, only show the part of the name that precedes the dot, so it can be hard to figure out the full file name.

The situation is even more difficult when the file exists somewhere other than than the current directory. File processing programs might be used on files that are stored literally anywhere in secondary memory. In order to locate these far-flung files, we must specify the complete path to locate the file in the user's computer system. The exact form of a path differs from system to system. On a Windows system, the complete file name with path might look something like this:

```
C:/users/susan/Documents/Python_Programs/users.txt
```

Not only is this a lot to type, but most users probably don't even know how to figure out the complete path+filename for any given file on their systems.

The solution to this problem is to allow users to browse the file system visually and navigate their way to a particular directory/file. Asking a user for a file name either for opening or saving is a common task across many applications, and the underlying operating system generally provides a standard/familiar way of doing this. The usual technique incorporates a dialog box (a special window for user interaction) that allows a user to click around in the file system using a mouse and either select or type in the name of a file. Fortunately for us, the `tkinter` GUI library included with (most) standard Python installations provides some simple-to-use functions that create dialog boxes for getting file names.

To ask the user for the name of a file to open, you can use the `askopenfilename` function. It is found in the `tkinter.filedialog` module. At the top of the program you will need to import the function:

```
from tkinter.filedialog import askopenfilename
```

The reason for the dot notation in the import is that `tkinter` is a package composed of multiple modules. In this case, we are specifying the `filedialog` module from `tkinter`. Rather than importing everything from this module, I specified just the one function that we are using here. Calling `askopenfilename` will pop up a system-appropriate file dialog box.

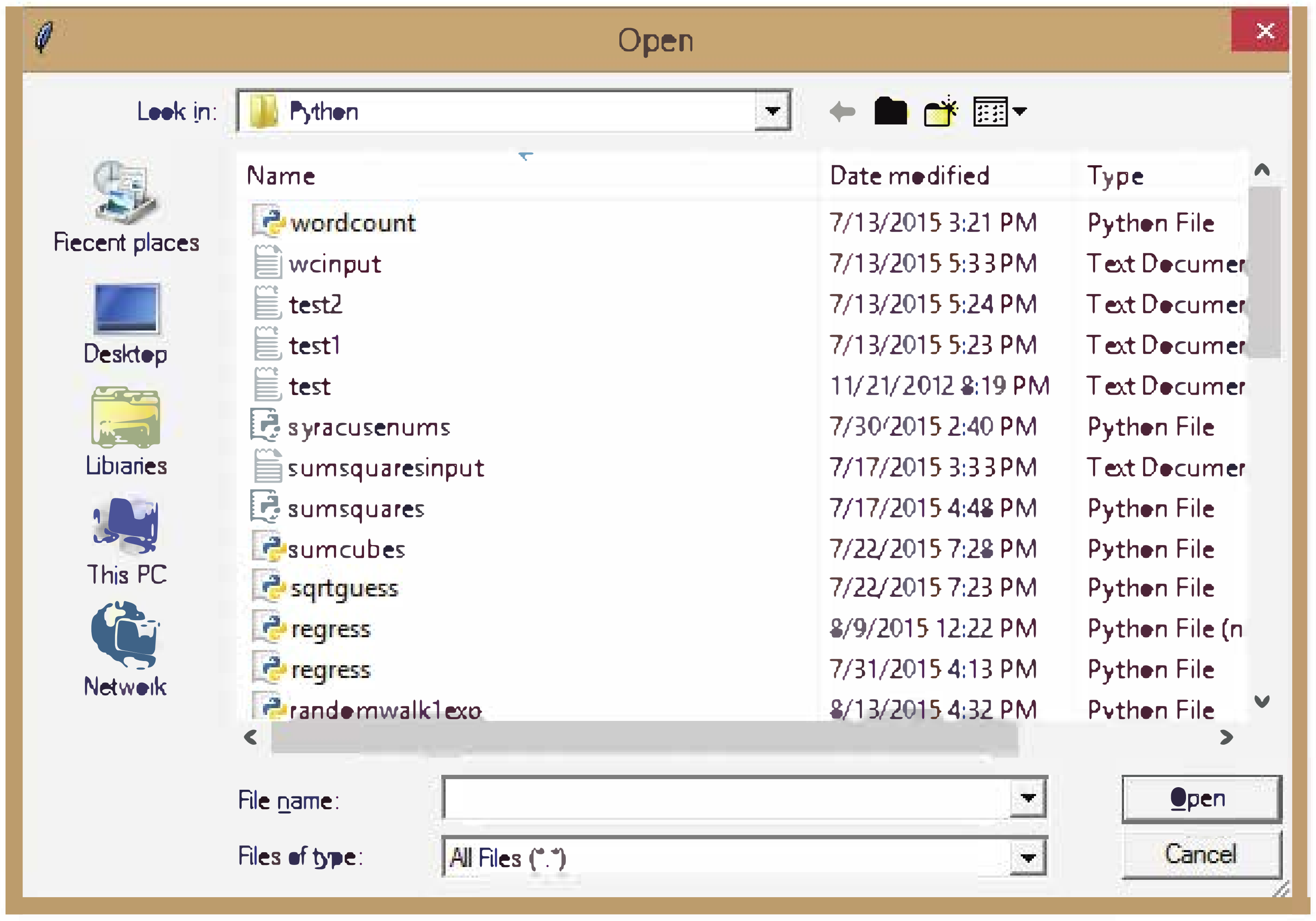
For example, to get the name of the user names file we could use a line of code like this:

```
infileName = askopenfilename()
```

The result of executing this line in Windows is shown in Figure 5.2. The dialog allows the user to either type in the name of the file or to simply select it with the mouse. When the user clicks the "Open" button, the complete path name of the file is returned as a string and saved into the variable `infileName`. If the user clicks the "Cancel" button, the function will simply return an empty string. In Chapter 7, you'll learn how you can test the resulting value and take different actions depending on which button the user selects.

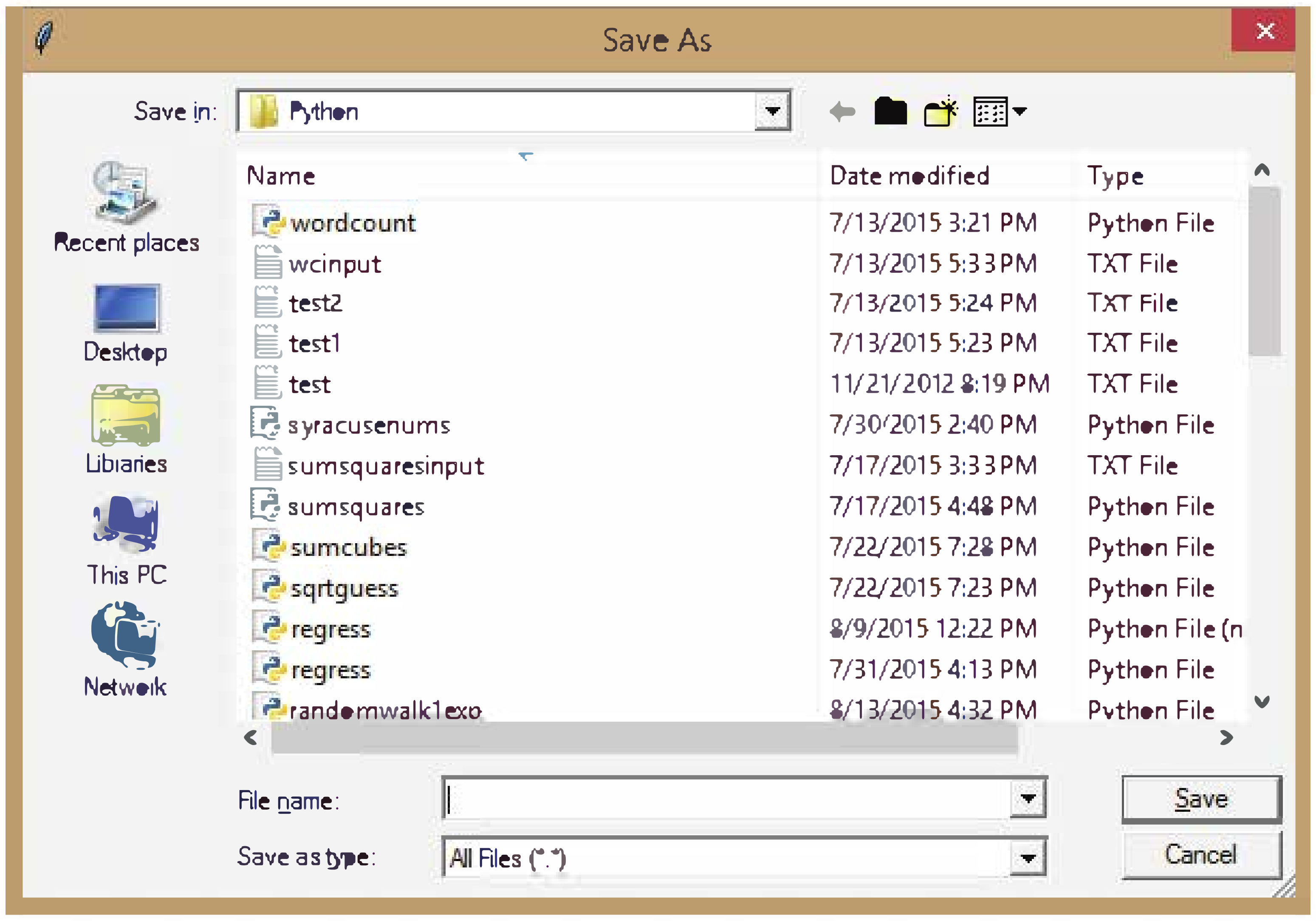
Python's `tkinter` provides an analogous function, `asksaveasfilename`, for saving files. It's usage is very similar.

```
from tkinter.filedialog import asksaveasfilename
...
outfileName = asksaveasfilename()
```



Name	Date modified	Type
wordcount	7/13/2015 3:21 PM	Python File
wcinput	7/13/2015 5:33 PM	Text Documen
test2	7/13/2015 5:24 PM	Text Documen
test1	7/13/2015 5:23 PM	Text Documen
test	11/21/2012 8:19 PM	Text Documen
syracusenums	7/30/2015 2:40 PM	Python File
sumsquaresinput	7/17/2015 3:33 PM	Text Documen
sumsquares	7/17/2015 4:48 PM	Python File
sumcubes	7/22/2015 7:28 PM	Python File
sqrtguess	7/22/2015 7:23 PM	Python File
regress	8/9/2015 12:22 PM	Python File (n
regress	7/31/2015 4:13 PM	Python File
randomwalk1exo	8/13/2015 4:32 PM	Python File

Figure 5.2: File dialog box from **askopenfilename**



Name	Date modified	Type
wordcount	7/13/2015 3:21 PM	Python File
wcinput	7/13/2015 5:33 PM	TXT File
test2	7/13/2015 5:24 PM	TXT File
test1	7/13/2015 5:23 PM	TXT File
test	11/21/2012 8:19 PM	TXT File
syracusenums	7/30/2015 2:40 PM	Python File
sumsquaresinput	7/17/2015 3:33 PM	TXT File
sumsquares	7/17/2015 4:48 PM	Python File
sumcubes	7/22/2015 7:28 PM	Python File
sqrtguess	7/22/2015 7:23 PM	Python File
regress	8/9/2015 12:22 PM	Python File (n
regress	7/31/2015 4:13 PM	Python File
randomwalk1exo	8/13/2015 4:32 PM	Python File

Figure 5.3: File dialog box from **asksaveasfilename**

An example dialog box for `asksaveasfilename` is shown in Figure 5.3. You can, of course, import both of these functions at once with an import like:

```
from tkinter.filedialog import askopenfilename, asksaveasfilename
```

Both of these functions also have numerous optional parameters so that a program can customize the the resulting dialogs, for example by changing the title or suggesting a default file name. If you are interested in those details, you should consult the Python documentation.

5.10 Chapter Summary

This chapter has covered important elements of the Python string, list, and file objects. Here is a summary of the highlights:

- Strings are sequences of characters. String literals can be delimited with either single or double quotes.
- Strings and lists can be manipulated with the built-in sequence operations for concatenation (`+`), repetition (`*`), indexing (`[]`), slicing (`[:]`), and length (`len()`). A `for` loop can be used to iterate through the characters of a string, items in a list, or lines of a file.
- One way of converting numeric information into string information is to use a string or a list as a lookup table.
- Lists are more general than strings.
  - Strings are always sequences of characters, whereas lists can contain values of any type.
  - Lists are mutable, which means that items in a list can be modified by assigning new values.
- Strings are represented in the computer as numeric codes. ASCII and Unicode are compatible standards that are used for specifying the correspondence between characters and the underlying codes. Python provides the `ord` and `chr` functions for translating between Unicode codes and characters.
- Python string and list objects include many useful built-in methods for string and list processing.

- The process of encoding data to keep it private is called encryption. There are two different kinds of encryption systems: private key and public key.
- Program input and output often involve string processing. Python provides numerous operators for converting back and forth between numbers and strings. The string formatting method (`format`) is particularly useful for producing nicely formatted output.
- Text files are multi-line strings stored in secondary memory. A text file may be opened for reading or writing. When opened for writing, the existing contents of the file are erased. Python provides three file-reading methods: `read()`, `readline()`, and `readlines()`. It is also possible to iterate through the lines of a file with a `for` loop. Data is written to a file using the `print` function. When processing is finished, a file should be closed.

5.11 Exercises

Review Questions

True/False

1. A Python string literal is always enclosed in double quotes.
2. The last character of a string `s` is at position `len(s)-1`.
3. A string always contains a single line of text.
4. In Python `"4" + "5"` is `"45"`.
5. Python lists are mutable, but strings are not.
6. ASCII is a standard for representing characters using numeric codes.
7. The `split` method breaks a string into a list of substrings, and `join` does the opposite.
8. A substitution cipher is a good way to keep sensitive information secure.
9. The `add` method can be used to add an item to the end of a list.
10. The process of associating a file with an object in a program is called "reading" the file.

Multiple Choice

1. Accessing a single character out of a string is called:
   a) slicing b) concatenation c) assignment d) indexing
2. Which of the following is the same as `s[0:-1]`?
   a) `s[-1]` b) `s[:]` c) `s[:len(s)-1]` d) `s[0:len(s)]`
3. What function gives the Unicode value of a character?
   a) `ord` b) `ascii` c) `chr` d) `eval`
4. Which of the following can *not* be used to convert a string of digits into a number?
   a) `int` b) `float` c) `str` d) `eval`
5. A successor to ASCII that includes characters from (nearly) all written languages is
   a) TELLI b) ASCII++ c) Unicode d) ISO
6. Which string method converts all the characters of a string to upper case?
   a) `capitalize` b) `capwords` c) `uppercase` d) `upper`
7. The string "slots" that are filled in by the `format` method are marked by:
   a) `%` b) `$` c) `[]` d) `{}`
8. Which of the following is *not* a file-reading method in Python?
   a) `read` b) `readline` c) `readall` d) `readlines`
9. The term for a program that does its input and output with files is
   a) file-oriented b) multi-line c) batch d) lame
10. Before reading or writing to a file, a file object must be created via
    a) `open` b) `create` c) `File` d) `Folder`

Discussion

1. Given the initial statements:

   ```
   s1 = "spam"
   s2 = "ni!"
   ```

   Show the result of evaluating each of the following string expressions.

a) `"The Knights who say, " + s2`
b) `3 * s1 + 2 * s2`
c) `s1[1]`
d) `s1[1:3]`
e) `s1[2] + s2[:2]`
f) `s1 + s2[-1]`
g) `s1.upper()`
h) `s2.upper().ljust(4) * 3`

2. Given the same initial statements as in the previous problem, show a Python expression that could construct each of the following results by performing string operations on s1 and s2.

a) `"NI"`
b) `"ni!spamni!"`
c) `"Spam Ni!  Spam Ni!  Spam Ni!"`
d) `"spam"`
e) `["sp","m"]`
f) `"spm"`

3. Show the output that would be generated by each of the following program fragments:

a)
```
for ch in "aardvark":
    print(ch)
```
b)
```
for w in "Now is the winter of our discontent...".split():
    print(w)
```
c)
```
for w in "Mississippi".split("i"):
    print(w, end=" ")
```
d)
```
msg = ""
for s in "secret".split("e"):
    msg = msg + s
print(msg)
```
e)
```
msg = ""
for ch in "secret":
    msg = msg + chr(ord(ch)+1)
print(msg)
```

4. Show the string that would result from each of the following string formatting operations. If the operation is not legal, explain why.

    a) `"Looks like {1} and {0} for breakfast".format("eggs", "spam")`

    b) `"There is {0} {1} {2} {3}".format(1,"spam", 4, "you")`

    c) `"Hello {0}".format("Susan", "Computewell")`

    d) `"{0:0.2f} {0:0.2f}".format(2.3, 2.3468)`

    e) `"{7.5f} {7.5f}".format(2.3, 2.3468)`

    f) `"Time left {0:02}:{1:05.2f}".format(1, 37.374)`

    g) `"{1:3}".format("14")`

5. Explain why public key encryption is more useful for securing communications on the Internet than private (shared) key encryption.

Programming Exercises

1. As discussed in the chapter, string formatting could be used to simplify the `dateconvert2.py` program. Go back and redo this program making use of the string-formatting method.

2. A certain CS professor gives 5-point quizzes that are graded on the scale 5-A, 4-B, 3-C, 2-D, 1-F, 0-F. Write a program that accepts a quiz score as an input and prints out the corresponding grade.

3. A certain CS professor gives 100-point exams that are graded on the scale 90–100:A, 80–89:B, 70–79:C, 60–69:D, <60:F. Write a program that accepts an exam score as input and prints out the corresponding grade.

4. An *acronym* is a word formed by taking the first letters of the words in a phrase and making a word from them. For example, RAM is an acronym for "random access memory." Write a program that allows the user to type in a phrase and then outputs the acronym for that phrase. *Note*: The acronym should be all uppercase, even if the words in the phrase are not capitalized.

5. Numerologists claim to be able to determine a person's character traits based on the "numeric value" of a name. The value of a name is determined by summing up the values of the letters of the name where "a" is 1, "b" is 2, "c" is 3, up to "z" being 26. For example, the name "Zelle"

would have the value $26 + 5 + 12 + 12 + 5 = 60$ (which happens to be a very auspicious number, by the way). Write a program that calculates the numeric value of a single name provided as input.

6. Expand your solution to the previous problem to allow the calculation of a complete name such as "John Marvin Zelle" or "John Jacob Jingleheimer Smith." The total value is just the sum of the numeric values of all the names.

7. A Caesar cipher is a simple substitution cipher based on the idea of shifting each letter of the plaintext message a fixed number (called the key) of positions in the alphabet. For example, if the key value is 2, the word "Sourpuss" would be encoded as "Uqwtrwuu." The original message can be recovered by "reencoding" it using the negative of the key.

   Write a program that can encode and decode Caesar ciphers. The input to the program will be a string of plaintext and the value of the key. The output will be an encoded message where each character in the original message is replaced by shifting it *key* characters in the Unicode character set. For example, if `ch` is a character in the string and `key` is the amount to shift, then the character that replaces `ch` can be calculated as: `chr(ord(ch) + key)`.

8. One problem with the previous exercise is that it does not deal with the case when we "drop off the end" of the alphabet. A true Caesar cipher does the shifting in a circular fashion where the next character after "z" is "a." Modify your solution to the previous problem to make it circular. You may assume that the input consists only of letters and spaces. *Hint*: Make a string containing all the characters of your alphabet and use positions in this string as your code. You do not have to shift "z" into "a"; just make sure that you use a circular shift over the entire sequence of characters in your alphabet string.

9. Write a program that counts the number of words in a sentence entered by the user.

10. Write a program that calculates the average word length in a sentence entered by the user.

11. Write an improved version of the `chaos.py` program from Chapter 1 that allows a user to input two initial values and the number of iterations,

and then prints a nicely formatted table showing how the values change over time. For example, if the starting values were .25 and .26 with 10 iterations, the table might look like this:

```
index    0.25         0.26
----------------------------
  1     0.731250     0.750360
  2     0.766441     0.730547
  3     0.698135     0.767707
  4     0.821896     0.695499
  5     0.570894     0.825942
  6     0.955399     0.560671
  7     0.166187     0.960644
  8     0.540418     0.147447
  9     0.968629     0.490255
 10     0.118509     0.974630
```

12. Write an improved version of the `futval.py` program from Chapter 2. Your program will prompt the user for the amount of the investment, the annualized interest rate, and the number of years of the investment. The program will then output a nicely formatted table that tracks the value of the investment year by year. Your output might look something like this:

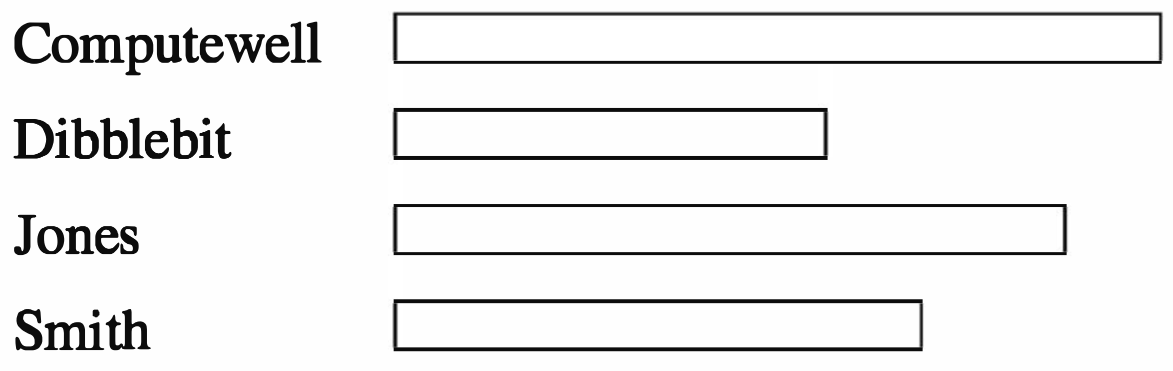
```
Year     Value
----------------
  0      $2000.00
  1      $2200.00
  2      $2420.00
  3      $2662.00
  4      $2928.20
  5      $3221.02
  6      $3542.12
  7      $3897.43
```

13. Redo any of the previous programming problems to make them batch-oriented (using text files for input and output).

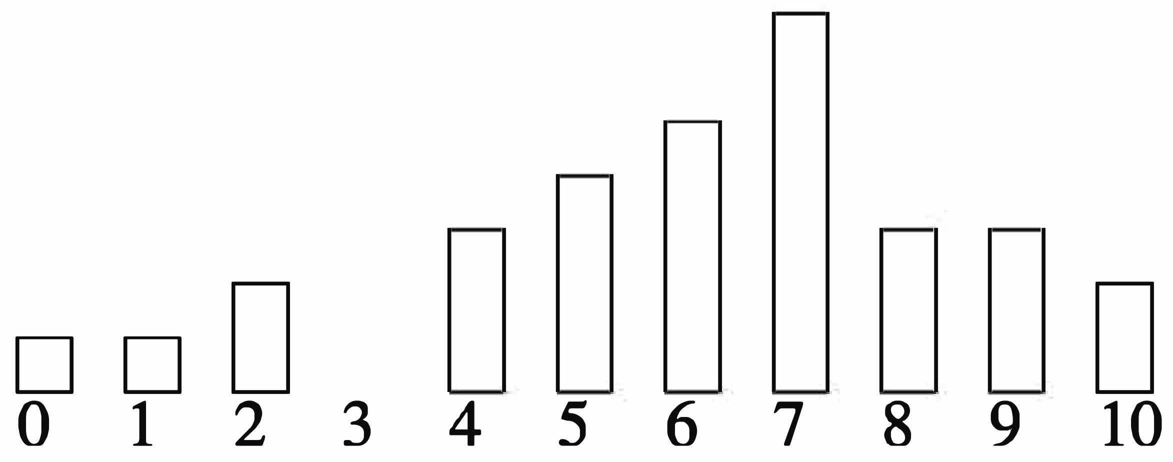
14. Word Count. A common utility on Unix/Linux systems is a small program called "wc." This program analyzes a file to determine the number of

lines, words, and characters contained therein. Write your own version of wc. The program should accept a file name as input and then print three numbers showing the count of lines, words, and characters in the file.

15. Write a program to plot a horizontal bar chart of student exam scores. Your program should get input from a file. The first line of the file contains the count of the number of students in the file, and each subsequent line contains a student's last name followed by a score in the range 0–100. Your program should draw a horizontal rectangle for each student where the length of the bar represents the student's score. The bars should all line up on their left-hand edges. *Hint*: Use the number of students to determine the size of the window and its coordinates. Bonus: label the bars at the left end with the students' names.



16. Write a program to draw a quiz score histogram. Your program should read data from a file. Each line of the file contains a number in the range 0–10. Your program must count the number of occurrences of each score and then draw a vertical bar chart with a bar for each possible score (0–10) with a height corresponding to the count of that score. For example, if 15 students got an 8, then the height of the bar for 8 should be 15. *Hint*: Use a list that stores the count for each possible score. An example histogram is shown below:



Chapter 6 Defining Functions

Objectives

- To understand why programmers divide programs up into sets of cooperating functions.
- To be able to define new functions in Python.
- To understand the details of function calls and parameter passing in Python.
- To write programs that use functions to reduce code duplication and increase program modularity.

6.1 The Function of Functions

The programs that we have written so far comprise a single function, usually called `main`. We have also been using pre-written functions and methods including built-in Python functions (e.g., `print`, `abs`), functions and methods from the Python standard libraries (e.g., `math.sqrt`), and methods from the `graphics` module (e.g., `myPoint.getX()`). Functions are an important tool for building sophisticated programs. This chapter covers the whys and hows of designing your own functions to make your programs easier to write and understand.

In Chapter 4, we looked at a graphic solution to the future value problem. Recall that this program makes use of the `graphics` library to draw a bar chart showing the growth of an investment. Here is the program as we left it:

```
# futval_graph2.py

from graphics import *
```

```
def main():
    # Introduction
    print("This program plots the growth of a 10-year investment.")

    # Get principal and interest rate
    principal = float(input("Enter the initial principal: "))
    apr = float(input("Enter the annualized interest rate: "))

    # Create a graphics window with labels on left edge
    win = GraphWin("Investment Growth Chart", 320, 240)
    win.setBackground("white")
    win.setCoords(-1.75,-200, 11.5, 10400)
    Text(Point(-1, 0), ' 0.0K').draw(win)
    Text(Point(-1, 2500), ' 2.5K').draw(win)
    Text(Point(-1, 5000), ' 5.0K').draw(win)
    Text(Point(-1, 7500), ' 7.5k').draw(win)
    Text(Point(-1, 10000), '10.0K').draw(win)

    # Draw bar for initial principal
    bar = Rectangle(Point(0, 0), Point(1, principal))
    bar.setFill("green")
    bar.setWidth(2)
    bar.draw(win)

    # Draw a bar for each subsequent year
    for year in range(1, 11):
        principal = principal * (1 + apr)
        bar = Rectangle(Point(year, 0), Point(year+1, principal))
        bar.setFill("green")
        bar.setWidth(2)
        bar.draw(win)

    input("Press <Enter> to quit.")
    win.close()

main()
```

This is certainly a workable program, but there is a nagging issue of program style that really should be addressed. Notice that this program draws bars in two different places. The initial bar is drawn just before the loop, and the subsequent bars are drawn inside the loop.

Having similar code like this in two places has some drawbacks. Obviously, one issue is having to write the code twice. A more subtle problem is that the code has to be maintained in two different places. Should we decide to change the color or other facets of the bars, we would have to make sure these changes occur in both places. Failing to keep related parts of the code in sync is a common problem in program maintenance.

Functions can be used to reduce code duplication and to make programs more understandable and easier to maintain. Before fixing up the future value program, let's take look at what functions have to offer.

6.2 Functions, Informally

You can think of a function as a *subprogram*—a small program inside a program. The basic idea of a function is that we write a sequence of statements and give that sequence a name. The instructions can then be executed at any point in the program by referring to the function name.

The part of the program that creates a function is called a *function definition*. When a function is subsequently used in a program, we say that the definition is *called* or *invoked*. A single function definition may be called at many different points of a program.

Let's take a concrete example. Suppose you want to write a program that prints out the lyrics to the "Happy Birthday" song. The standard lyrics look like this:

```
Happy birthday to you!
Happy birthday to you!
Happy birthday, dear <insert-name>.
Happy birthday to you!
```

We're going to play with this example in the interactive Python environment. You might want to fire up Python and try some of this out yourself.

A simple approach to this problem is to use four `print` statements. Here's an interactive session that creates a program for singing "Happy Birthday" to Fred.

```
>>> def main():
        print("Happy birthday to you!")
```

```
    print("Happy birthday to you!")
    print("Happy birthday, dear Fred.")
    print("Happy birthday to you!")
```

We can then run this program to get our lyrics:

```
>>> main()
Happy birthday to you!
Happy birthday to you!
Happy birthday, dear Fred.
Happy birthday to you!
```

Obviously, there is some duplicated code in this program. For such a simple program, that's not a big deal, but even here it's a bit annoying to keep retyping the same line. Let's introduce a function that prints the lyrics of the first, second, and fourth lines.

```
>>> def happy():
        print("Happy birthday to you!")
```

We have defined a new function called `happy`. Here is an example of what it does:

```
>>> happy()
Happy birthday to you!
```

Invoking the `happy` command causes Python to print a line of the song.

Now we can redo the verse for Fred using `happy`. Let's call our new version `singFred`.

```
>>> def singFred():
        happy()
        happy()
        print("Happy birthday, dear Fred.")
        happy()
```

This version required much less typing, thanks to the `happy` command. Let's try printing the lyrics for Fred just to make sure it works.

```
>>> singFred()
Happy birthday to you!
Happy birthday to you!
Happy birthday, dear Fred.
Happy birthday to you!
```

So far, so good. Now suppose that it's also Lucy's birthday, and we want to sing a verse for Fred followed by a verse for Lucy. We've already got the verse for Fred; we can prepare one for Lucy as well.

```
>>> def singLucy():
        happy()
        happy()
        print("Happy birthday, dear Lucy.")
        happy()
```

Now we can write a `main` program that sings to both Fred and Lucy:

```
>>> def main():
        singFred()
        print()
        singLucy()
```

The bare `print` between the two function calls puts a space between the verses in our output. And here's the final product in action:

```
>>> main()
Happy birthday to you!
Happy birthday to you!
Happy birthday, dear Fred.
Happy birthday to you!

Happy birthday to you!
Happy birthday to you!
Happy birthday, dear Lucy.
Happy birthday to you!
```

Well now, that certainly seems to work, and we've removed some of the duplication by defining the `happy` function. However, something still doesn't feel quite right. We have two functions, `singFred` and `singLucy`, that are almost identical. Following this approach, adding a verse for Elmer would have us create a `singElmer` function that looks just like those for Fred and Lucy. Can't we do something about the proliferation of verses?

Notice that the only difference between `singFred` and `singLucy` is the name at the end of the third `print` statement. The verses are exactly the same except for this one changing part. We can collapse these two functions together by using a *parameter*. Let's write a generic function called `sing`:

```
>>> def sing(person):
        happy()
        happy()
        print("Happy Birthday, dear", person + ".")
        happy()
```

This function makes use of a parameter named `person`. A parameter is a variable that is initialized when the function is called. We can use the `sing` function to print a verse for either Fred or Lucy. We just need to supply the name as a parameter when we invoke the function:

```
>>> sing("Fred")
Happy birthday to you!
Happy birthday to you!
Happy Birthday, dear Fred.
Happy birthday to you!

>>> sing("Lucy")
Happy birthday to you!
Happy birthday to you!
Happy Birthday, dear Lucy.
Happy birthday to you!
```

Let's finish with a program that sings to all three of our birthday people:

```
>>> def main():
        sing("Fred")
        print()
        sing("Lucy")
        print()
        sing("Elmer")
```

It doesn't get much easier than that.

Here is the complete program as a module file:

```
# happy.py

def happy():
    print("Happy Birthday to you!")
```

```
def sing(person):
    happy()
    happy()
    print("Happy birthday, dear", person + ".")
    happy()

def main():
    sing("Fred")
    print()
    sing("Lucy")
    print()
    sing("Elmer")

main()
```

6.3 Future Value with a Function

Now that you've seen how defining functions can help solve the code duplication problem, let's return to the future value graph. Remember, the problem is that bars of the graph are drawn at two different places in the program. The code just before the loop looks like this:

```
# Draw bar for initial principal
bar = Rectangle(Point(0, 0), Point(1, principal))
bar.setFill("green")
bar.setWidth(2)
bar.draw(win)
```

And the code inside the loop is as follows:

```
bar = Rectangle(Point(year, 0), Point(year+1, principal))
bar.setFill("green")
bar.setWidth(2)
bar.draw(win)
```

Let's try to combine these two into a single function that draws a bar on the screen.

In order to draw the bar, we need some information. Specifically, we need to know what year the bar will be for, how tall the bar will be, and what window the bar will be drawn in. These three values will be supplied as parameters for the function. Here's the function definition:

```
def drawBar(window, year, height):
    # Draw a bar in window for given year with given height
    bar = Rectangle(Point(year, 0), Point(year+1, height))
    bar.setFill("green")
    bar.setWidth(2)
    bar.draw(window)
```

To use this function, we just need to supply values for the three parameters. For example, if `win` is a `GraphWin`, we can draw a bar for year 0 and a principal of $2,000 by invoking `drawBar` like this:

```
drawBar(win, 0, 2000)
```

Incorporating the `drawBar` function, here is the latest version of our future value program:

```
# futval_graph3.py
from graphics import *

def drawBar(window, year, height):
    # Draw a bar in window starting at year with given height
    bar = Rectangle(Point(year, 0), Point(year+1, height))
    bar.setFill("green")
    bar.setWidth(2)
    bar.draw(window)

def main():
    # Introduction
    print("This program plots the growth of a 10-year investment.")

    # Get principal and interest rate
    principal = float(input("Enter the initial principal: "))
    apr = float(input("Enter the annualized interest rate: "))

    # Create a graphics window with labels on left edge
    win = GraphWin("Investment Growth Chart", 320, 240)
    win.setBackground("white")
    win.setCoords(-1.75,-200, 11.5, 10400)
    Text(Point(-1, 0), ' 0.0K').draw(win)
    Text(Point(-1, 2500), ' 2.5K').draw(win)
    Text(Point(-1, 5000), ' 5.0K').draw(win)
    Text(Point(-1, 7500), ' 7.5k').draw(win)
    Text(Point(-1, 10000), '10.0K').draw(win)

    drawBar(win, 0, principal)
    for year in range(1, 11):
        principal = principal * (1 + apr)
        drawBar(win, year, principal)

    input("Press <Enter> to quit.")
    win.close()
main()
```

You can see how drawBar has eliminated the duplicated code. Should we wish to change the appearance of the bars in the graph, we only need to change the code in one spot, the definition of drawBar. Don't worry yet if you don't understand every detail of this example. You still have some things to learn about functions.

6.4 Functions and Parameters: The Exciting Details

You may be wondering about the choice of parameters for the drawBar function. Obviously, the year for which a bar is being drawn and the height of the bar are the changeable parts in the drawing of a bar. But why is window also a parameter to this function? After all, we will be drawing all of the bars in the same window; it doesn't seem to change.

The reason for making window a parameter has to do with the *scope* of variables in function definitions. Scope refers to the places in a program where a given variable may be referenced. Remember, each function is its own little subprogram. The variables used inside one function are *local* to that function, even if they happen to have the same name as variables that appear inside another function.

The only way for a function to see a variable from another function is for that variable to be passed as a parameter.[1] Since the GraphWin (assigned to

[1]Technically, it is possible to reference a variable from a function that is nested inside another function, but function nesting is beyond the scope of this discussion.

the variable win) is created inside main, it is not directly accessible in drawBar. However, the window parameter in drawBar gets assigned the value of win from main when drawBar is called. To see how this happens, we need to take a more detailed look at the function invocation process.

A function definition looks like this:

```
def <name>(<formal-parameters>):
    <body>
```

The name of the function must be an identifier, and formal-parameters is a (possibly empty) sequence of variable names (also identifiers). The formal parameters, like all variables used in the function, are only accessible in the body of the function. Variables with identical names elsewhere in the program are distinct from the formal parameters and variables inside the function body.

A function is called by using its name followed by a list of *actual parameters* or *arguments*.

```
<name>(<actual-parameters>)
```

When Python comes to a function call, it initiates a four-step process:

1. The calling program suspends execution at the point of the call.
2. The formal parameters of the function get assigned the values supplied by the actual parameters in the call.
3. The body of the function is executed.
4. Control returns to the point just after where the function was called.

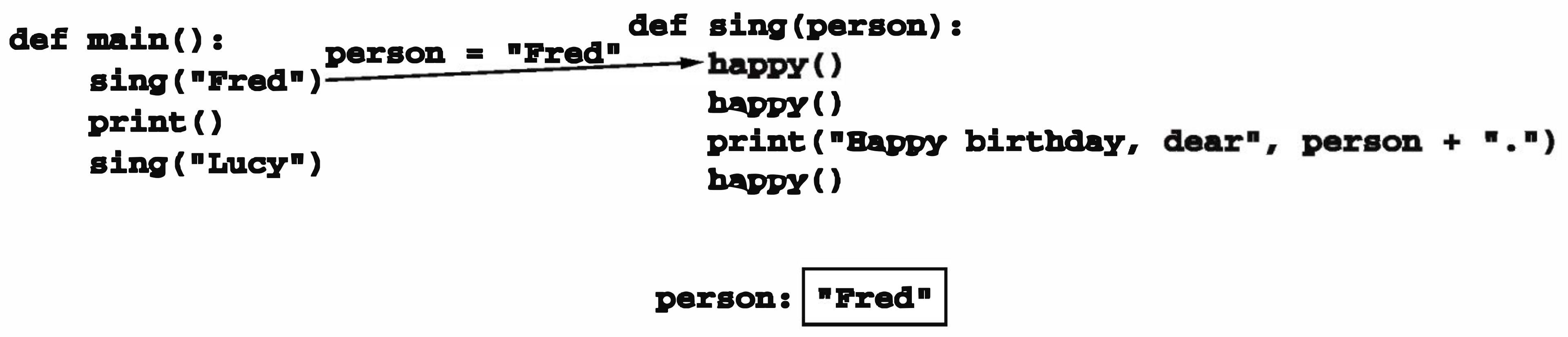
Returning to the Happy Birthday example, let's trace through the singing of two verses. Here is part of the body from main:

```
sing("Fred")
print()
sing("Lucy")
```

When Python gets to sing("Fred"), execution of main is temporarily suspended. At this point, Python looks up the definition of sing and sees that it has a single formal parameter, person. The formal parameter is assigned the value of the actual parameter, so it is as if we had executed this statement:

```
person = "Fred"
```

A snapshot of the situation is shown in Figure 6.1. Notice the variable person inside sing has just been initialized.

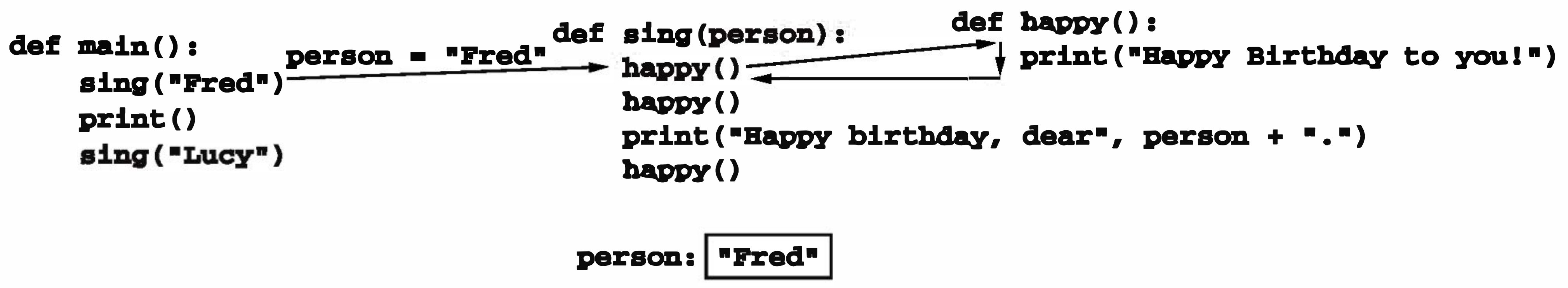


```
def main():                                def sing(person):
    sing("Fred")   person = "Fred"  ---->      happy()
    print()                                    happy()
    sing("Lucy")                               print("Happy birthday, dear", person + ".")
                                               happy()

                          person: "Fred"
```

Figure 6.1: Illustration of control transferring to sing

At this point, Python begins executing the body of sing. The first statement is another function call, this one to happy. Python suspends execution of sing and transfers control to the called function. The body of happy consists of a single print. This statement is executed, and then control returns to where it left off in sing. Figure 6.2 shows a snapshot of the execution so far.



```
def main():                             def sing(person):        def happy():
    sing("Fred")  person = "Fred" --->      happy()  ------->        print("Happy Birthday to you!")
    print()                                 happy()
    sing("Lucy")                            print("Happy birthday, dear", person + ".")
                                            happy()

                       person: "Fred"
```

Figure 6.2: Snapshot of completed call to happy

Execution continues in this manner with Python making two more side trips back to happy to complete the execution of sing. When Python gets to the end of sing, control then returns to main and continues immediately after the function call. Figure 6.3 shows where we are at that point. Notice that the person variable in sing has disappeared. The memory occupied by local function variables is reclaimed when the function finishes. Local variables do not retain any values from one function execution to the next.

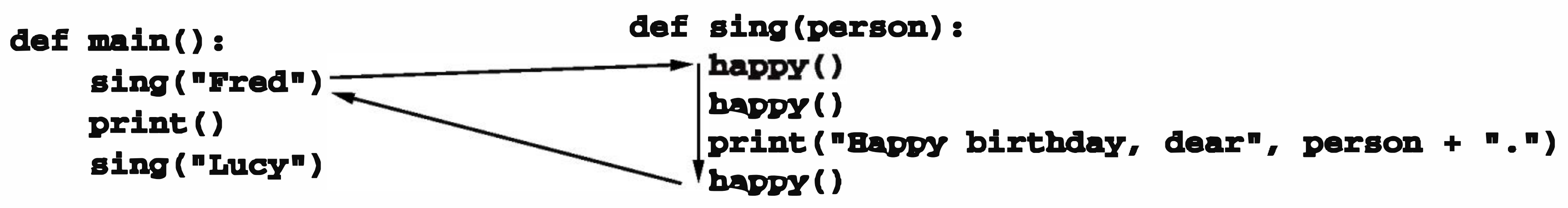


Figure 6.3: Snapshot of completed call to `sing`

The next statement to execute is the bare `print` statement in `main`. This produces a blank line in the output. Then Python encounters another call to `sing`. As before, control transfers to the function definition. This time the formal parameter is `"Lucy"`. Figure 6.4 shows the situation as `sing` begins to execute for the second time.

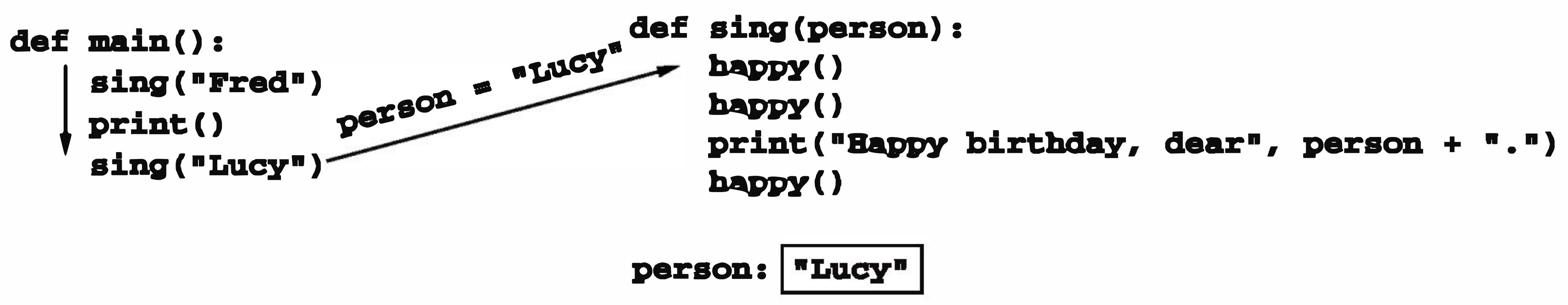


Figure 6.4: Snapshot of second call to `sing`

Now we'll fast forward to the end. The function body of `sing` is executed for Lucy (with three side trips through `happy`) and control returns to `main` just after the point of the function call. Now we have reached the bottom of our code fragment, as illustrated by Figure 6.5. These three statements in `main` have caused `sing` to execute twice and `happy` to execute six times. Overall, nine total lines of output were generated.

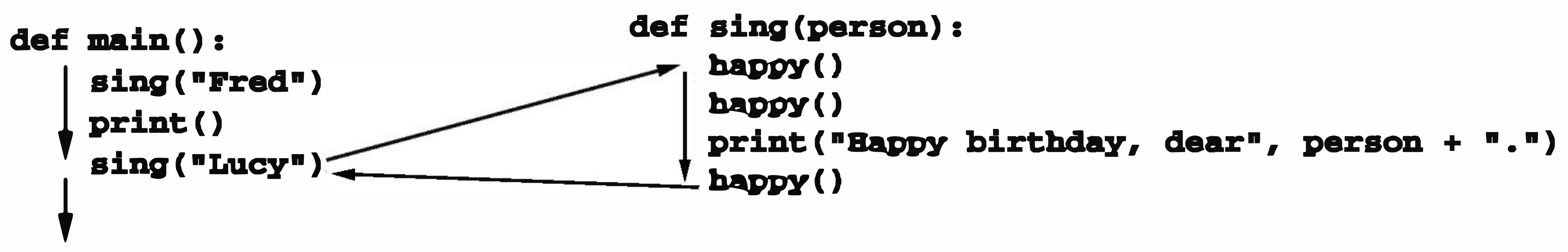


Figure 6.5: Completion of second call to `sing`

Hopefully you're getting the hang of how function calls work. One point that this example did not address is the use of multiple parameters. Usually when a function definition has several parameters, the actual parameters are matched up with the formal parameters by *position*. The first actual parameter is assigned to the first formal parameter, the second actual is assigned to the second formal, etc. It's possible to modify this behavior using keyword parameters, which are matched up by name (e.g., `end=""` in a call to `print`). However, we will rely on positional matching for all of our example functions.

As an example, look again at the use of the `drawBar` function from the future value program. Here is the call to draw the initial bar:

```
drawBar(win, 0, principal)
```

When Python transfers control to `drawBar`, these parameters are matched up to the formal parameters in the function heading:

```
def drawBar(window, year, height):
```

The net effect is as if the function body had been prefaced with three assignment statements:

```
window = win
year = 0
height = principal
```

You must always be careful when calling a function that you get the actual parameters in the correct order to match the function definition.

6.5 Functions That Return Values

You have seen that parameter passing provides a mechanism for initializing the variables in a function. In a way, parameters act as inputs to a function. We can call a function many times and get different results by changing the input parameters. Oftentimes we also want to get information back out of a function. In fact, the fundamental ideas and vocabulary of functions are borrowed from mathematics, where a function is considered to be a relation between input variables and output variables. For example, a mathematician might define a function, f, that computes the square of its input. Mathematically we would write something like this:

$$f(x) = x^2$$

This shows that f is a function that operates on a single variable (named x here) and produces a value that is the square of x.

As with Python functions, mathematicians use parenthetical notation to show the application of a function. For example, $f(5) = 25$ states the fact that when f is applied to 5, the result is 25. We would say "f of 5 equals 25." Mathematical functions are not restricted to a single argument. We might, for example, define a function that uses the Pythagorean Theorem to produce the length of the hypotenuse of a right triangle given the lengths of the legs. Let's call the function h.

$$h(x, y) = \sqrt{x^2 + y^2}$$

From this definition, you should be able to verify that $h(3, 4) = 5$.

So far we have been discussing Python function details with examples where functions are being used as new commands and functions are invoked to carry out the commands. But in the mathematical view, function calls are really expressions that produce a result. We can easily extend our view of Python functions to accomodate this idea. In fact, you have already seen numerous examples of this type of function. For example, consider this call to the `sqrt` function from the `math` library:

```
discRt = math.sqrt(b*b - 4*a*c)
```

Here the value of `b*b - 4*a*c` is the actual parameter of the `math.sqrt` function. Since the function call occurs on the right side of an assignment statement, that means it is an expression. The `math.sqrt` function produces a value that is then assigned to the variable `discRt`. Technically, we say that `sqrt` *returns* the square root of its argument.

It's very easy to write functions that return values. Here's a Python implementation of a function that returns the square of its argument:

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

Do you see how this function definition is very similar to the mathematical version ($f(x)$) above? The body of the Python function consists of a single `return` statement. When Python encounters a `return`, it immediately exits the current function and returns control to the point just after where the function was called. In addition, the value provided in the `return` statement is sent back to the caller as an expression result. Essentially, this just adds one small detail to the four-step function call process outlined before: the return value from a function is used as the expression result.

The effect is that we can use our `square` function any place in our code that an expression would be legal. Here are some interactive examples:

```
>>> square(3)
9
>>> print(square(4))
16
>>> x = 5
>>> y = square(x)
>>> print(y)
25
>>> print(square(x) + square(3))
34
```

Let's use the square function to write another function, one that finds the distance between two points. Given two points (x_1, y_1) and (x_2, y_2), the distance between them is calculated as $\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$. Here is a Python function to compute the distance between two `Point` objects:

```
def distance(p1, p2):
    dist = math.sqrt(square(p2.getX() - p1.getX())
                     + square(p2.getY() - p1.getY())
    return dist
```

Using the `distance` function, we can augment the interactive triangle program from Chapter 4 to calculate the perimeter of the triangle. Here's the complete program:

```
# Program: triangle2.py
import math
from graphics import *

def square(x):
    return x ** 2

def distance(p1, p2):
    dist = math.sqrt(square(p2.getX() - p1.getX())
                     + square(p2.getY() - p1.getY()))
    return dist

def main():
    win = GraphWin("Draw a Triangle")
    win.setCoords(0.0, 0.0, 10.0, 10.0)
    message = Text(Point(5, 0.5), "Click on three points")
    message.draw(win)

    # Get and draw three vertices of triangle
    p1 = win.getMouse()
    p1.draw(win)
    p2 = win.getMouse()
    p2.draw(win)
    p3 = win.getMouse()
    p3.draw(win)

    # Use Polygon object to draw the triangle
    triangle = Polygon(p1,p2,p3)
    triangle.setFill("peachpuff")
    triangle.setOutline("cyan")
    triangle.draw(win)

    # Calculate the perimeter of the triangle
    perim = distance(p1,p2) + distance(p2,p3) + distance(p3,p1)
    message.setText("The perimeter is: {0:0.2f}".format(perim))

    # Wait for another click to exit
    win.getMouse()
    win.close()

main()
```

You can see how `distance` is called three times in one line to compute the perimeter of the triangle. Using a function here saves quite a bit of tedious coding. Value-returning functions are extremely useful and flexible because they can be combined in expressions like this.

By the way, the order of the function definitions in the program is not important. It would have worked just the same with the `main` function defined at the top, for example. We just have to make sure a function is defined before the program actually tries to run it. Since the call to `main()` does not happen until the very last line of the module, all of the functions will be defined before the program actually starts running.

As another example, let's go back to the Happy Birthday program. In the original version, we used several functions containing `print` statements. Rather than having our helper functions do the printing, we could simply have them return values, strings in this case, that are then printed by `main`. Consider this version of the program:

```
# happy2.py

def happy():
    return "Happy Birthday to you!\n"

def verseFor(person):
    lyrics = happy()*2 + "Happy birthday, dear " + person + ".\n" + happy()
    return lyrics

def main():
    for person in ["Fred", "Lucy", "Elmer"]:
        print(verseFor(person))

main()
```

Notice that all the printing is carried out in one place (`main`) while `happy` and `verseFor` are just responsible for creating and returning appropriate strings. Through the magic of value-returning functions, we have streamlined the program so that an entire verse is built in a single string expression.

```
lyrics = happy()*2 + "Happy birthday, dear " + person + ".\n" + happy()
```

Make sure you carefully examine and understand this line of code; it really illustrates the power and beauty of value-returning functions.

In addition to being more elegant, this version of the program is also more flexible than the original because the printing is no longer distributed across multiple functions. For example, we can easily modify the program to write the results into a file instead of to the screen. All we have to do is open a file for writing and add a `file=` parameter to the print statement. No revision of the other functions is required. Here's the complete modification:

```
def main():
    outf = open("Happy_Birthday.txt", "w")
    for person in ["Fred", "Lucy", "Elmer"]:
        print(verseFor(person), file=outf)
    outf.close()
```

In general, it's almost always better (more flexible, that is) to have functions return values rather than printing information to the screen. That way the caller can choose whether to print the information or put it to some other use.

Sometimes a function needs to return more than one value. This can be done by simply listing more than one expression in the `return` statement. As a silly example, here is a function that computes both the sum and the difference of two numbers:

```
def sumDiff(x,y):
    sum = x + y
    diff = x - y
    return sum, diff
```

As you can see, this `return` hands back two values. When calling this function, we would place it in a simultaneous assignment:

```
num1, num2 = input("Please enter two numbers (num1, num2) ").split(",")
s, d  = sumDiff(float(num1), float(num2))
print("The sum is", s, "and the difference is", d)
```

As with parameters, when multiple values are returned from a function, they are assigned to variables by position. In this example, `s` will get the first value listed in the `return` (`sum`), and `d` will get the second value (`diff`).

That's just about all there is to know about value-returning functions in Python. There is one "gotcha" to warn you about. Technically, all functions in Python return a value, regardless of whether the function actually contains a `return` statement. Functions without a `return` always hand back a special object, denoted `None`. This object is often used as a sort of default value for variables that don't currently hold anything useful. A common mistake that new (and not-so-new) programmers make is writing what should be a value-returning function but forgetting to include a `return` statement at the end.

Suppose we forget to include the `return` statement at the end of the `distance` function:

```
def distance(p1, p2):
    dist = math.sqrt(square(p2.getX() - p1.getX())
                    + square(p2.getY() - p1.getY()))
```

Running the revised triangle program with this version of `distance` generates this Python error message:

```
Traceback (most recent call last):
  File "triangle2.py", line 42, in <module>
    main()
  File "triangle2.py", line 35, in main
    perim = distance(p1,p2) + distance(p2,p3) + distance(p3,p1)
TypeError: unsupported operand type(s) for +: 'NoneType' and 'NoneType'
```

The problem here is that this version of `distance` does not return a number; it always hands back the value `None`. Addition is not defined for `None` (which has the special type `NoneType`), and so Python complains. If your value-returning functions are producing strange error messages involving `None` or if your programs print out a mysterious "None" in the midst your output, check to see whether you missed a `return` statement.

6.6 Functions that Modify Parameters

Return values are the main way to send information from a function back to the part of the program that called the function. In some cases, functions can also communicate back to the calling program by making changes to the function parameters. Understanding when and how this is possible requires the mastery of some subtle details about how assignment works in Python and the effect this has on the relationship between the actual and formal parameters used in a function call.

Let's start with a simple example. Suppose you are writing a program that manages bank accounts or investments. One of the common tasks that must be performed is to accumulate interest on an account (as we did in the future value program). We might consider writing a function that automatically adds the interest to the account balance. Here is a first attempt at such a function:

```
# addinterest1.py
def addInterest(balance, rate):
    newBalance = balance * (1+rate)
    balance = newBalance
```

The intent of this function is to set the balance of the account to a value that has been updated by the amount of interest.

Let's try out our function by writing a very small test program:

```
def test():
    amount = 1000
    rate = 0.05
    addInterest(amount, rate)
    print(amount)
```

What do you think this program will print? Our intent is that 5% should be added to amount, giving a result of 1050. Here's what actually happens:

```
>>>test()
1000
```

As you can see, amount is unchanged! What has gone wrong?

Actually, nothing has gone wrong. If you consider carefully what we have discussed so far regarding functions and parameters, you will see that this is exactly the result that we should expect. Let's trace the execution of this example to see what happens. The first two lines of the test function create two local variables called amount and rate which are given the initial values of 1000 and 0.05, respectively.

Next, control transfers to the addInterest function. The formal parameters balance and rate are assigned the values from the actual parameters amount and rate. Remember, even though the name rate appears in both functions, these are two separate variables. The situation as addInterest begins to execute is shown in Figure 6.6. Notice that the assignment of parameters causes the variables balance and rate in addInterest to refer to the *values* of the actual parameters.

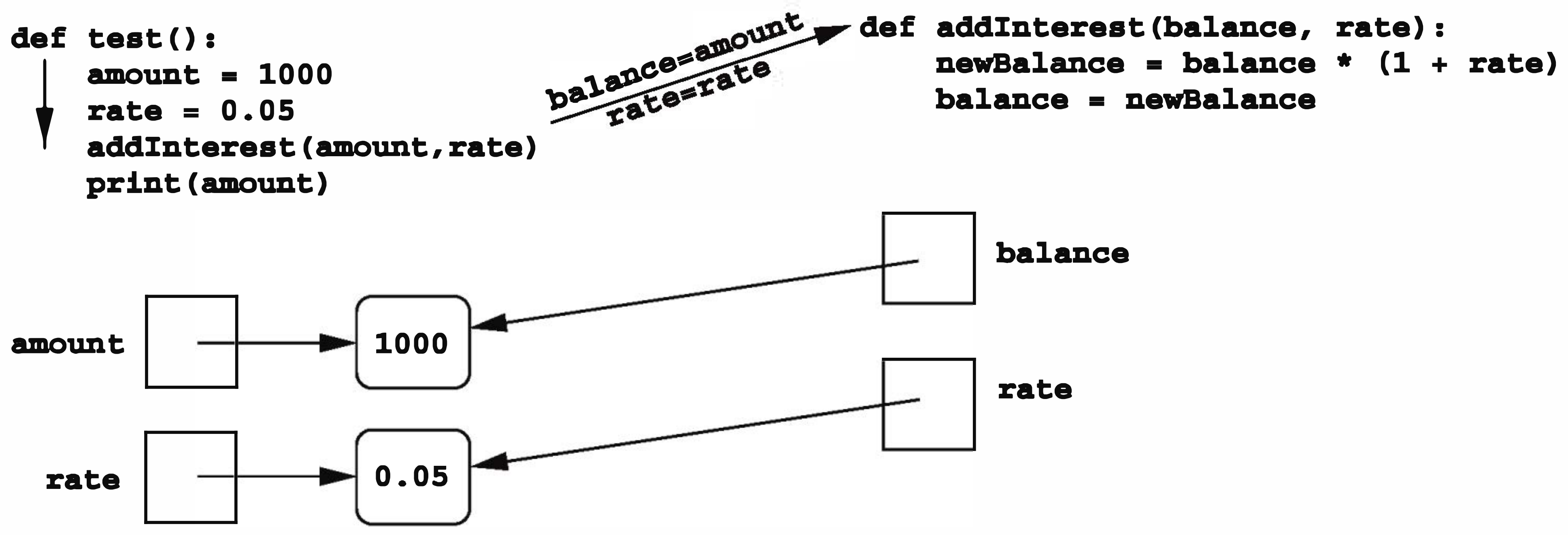


Figure 6.6: Transfer of control to addInterest

Executing the first line of `addInterest` creates a new variable, `newBalance`. Now comes the key step. The next statement in `addInterest` assigns `balance` to have the same value as `newBalance`. The result is shown in Figure 6.7. Notice that `balance` now refers to the same value as `newBalance`, but this *had no effect on* `amount` in the `test` function.

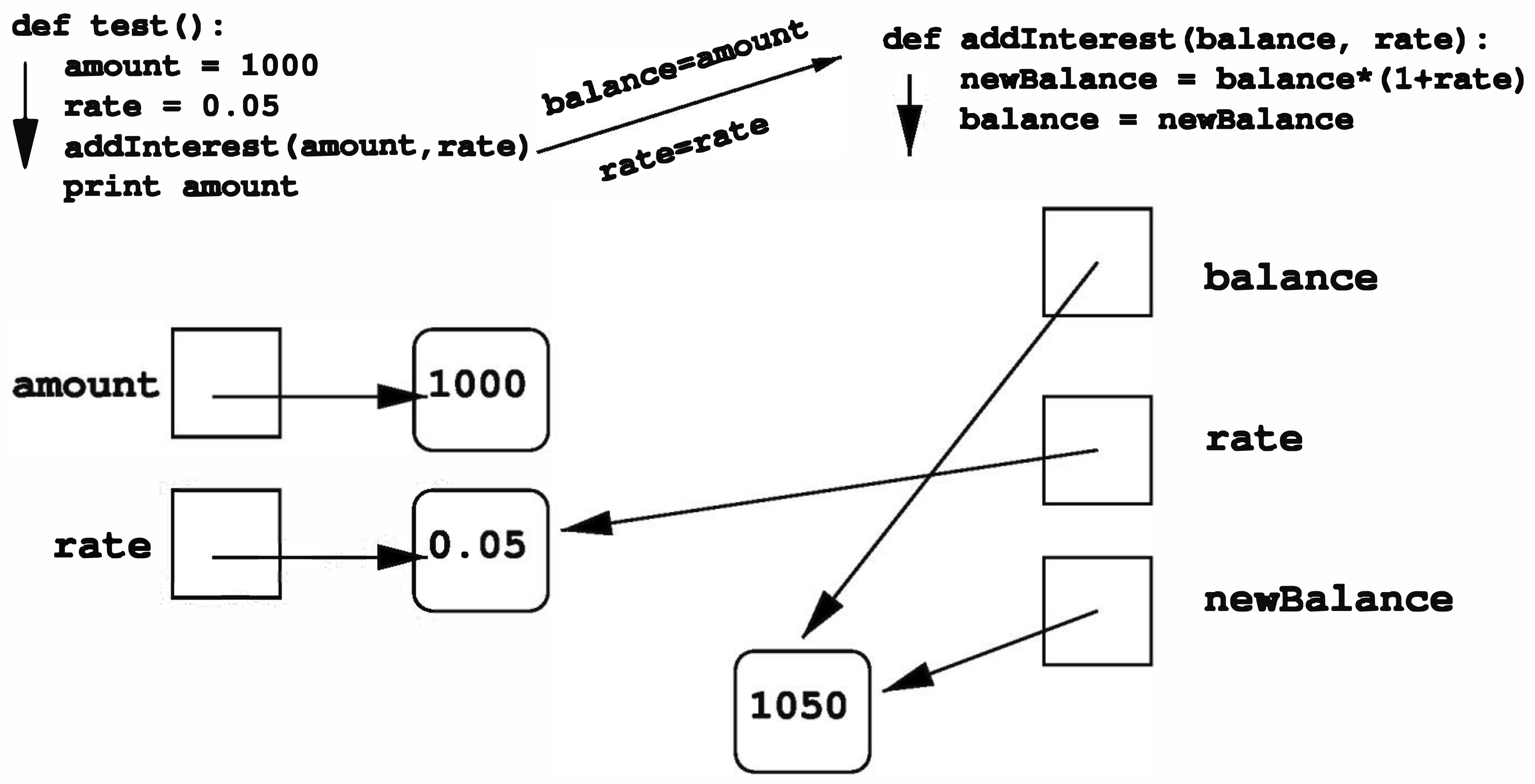


Figure 6.7: Assignment of `balance`

At this point, execution of `addInterest` has completed and control returns to `test`. The local variables (including parameters) in `addInterest` go away, but `amount` and `rate` in the `test` function still refer to the initial values of 1000 and 0.05, respectively. Of course, the program prints `amount` as 1000.

To summarize the situation, the formal parameters of a function only receive the *values* of the actual parameters. The function does not have access to the variable that holds the actual parameter; therefore, assigning a new value to a formal parameter has no effect on the variable containing the actual parameter. In programming language parlance, Python passes all parameters *by value*.

Some programming languages (e.g., C++ and Ada), do allow variables themselves to be sent as parameters to a function. Such a mechanism is called passing parameters *by reference*. When a variable is passed by reference, assigning a new value to the formal parameter actually changes the value of the parameter variable in the calling program.

Since Python does not allow passing parameters by reference, an obvious alternative is to change our `addInterest` function so that it returns the `newBalance`.

This value can then be used to update the `amount` in the `test` function. Here's a working version (`addinterest2.py`):

```
def addInterest(balance, rate):
    newBalance = balance * (1+rate)
    return newBalance

def test():
    amount = 1000
    rate = 0.05
    amount = addInterest(amount, rate)
    print(amount)
```

You should easily be able to trace through the execution of this program to see how we get this output:

```
>>>test()
1050
```

Now suppose instead of looking at a single account, we are writing a program that deals with many bank accounts. We could store the account balances in a Python list. It would be nice to have an `addInterest` function that adds the accrued interest to all of the balances in the list. If `balances` is a list of account balances, we can update the first amount in the list (the one at index 0) with a line of code like this:

```
balances[0] = balances[0] * (1 + rate)
```

Remember, this works because lists are mutable. This line of code essentially says, "multiply the value in the 0th position of the list by $(1 + rate)$ and store the result back into the 0th position of the list." Of course, a very similar line of code would work to update the balance of the next location in the list; we just replace the `0`s with `1`s:

```
balances[1] = balances[1] * (1 + rate)
```

A more general way of updating all the balances in a list is to use a loop that goes through positions $0, 1, \ldots, length - 1$. Consider `addinterest3.py`:

```
def addInterest(balances, rate):
    for i in range(len(balances)):
        balances[i] = balances[i] * (1+rate)
```

```
def test():
    amounts = [1000, 2200, 800, 360]
    rate = 0.05
    addInterest(amounts, rate)
    print(amounts)
```

Take a moment to study this program. The `test` function starts by setting `amounts` to be a list of four values. Then the `addInterest` function is called sending `amounts` as the first parameter. After the function call, the value of `amounts` is printed out. What do you expect to see? Let's run the program and see what happens:

```
>>> test()
[1050.0, 2310.0, 840.0, 378.0]
```

Isn't that interesting? In this example, the function seems to change the value of the `amounts` variable. But I just told you that Python passes parameters by value, so the variable itself (`amounts`) *can't* be changed by the function. So what's going on here?

The first two lines of `test` create the variables `amounts` and `rates`, and then control transfers to the `addInterest` function. The situation at this point is depicted in Figure 6.8.

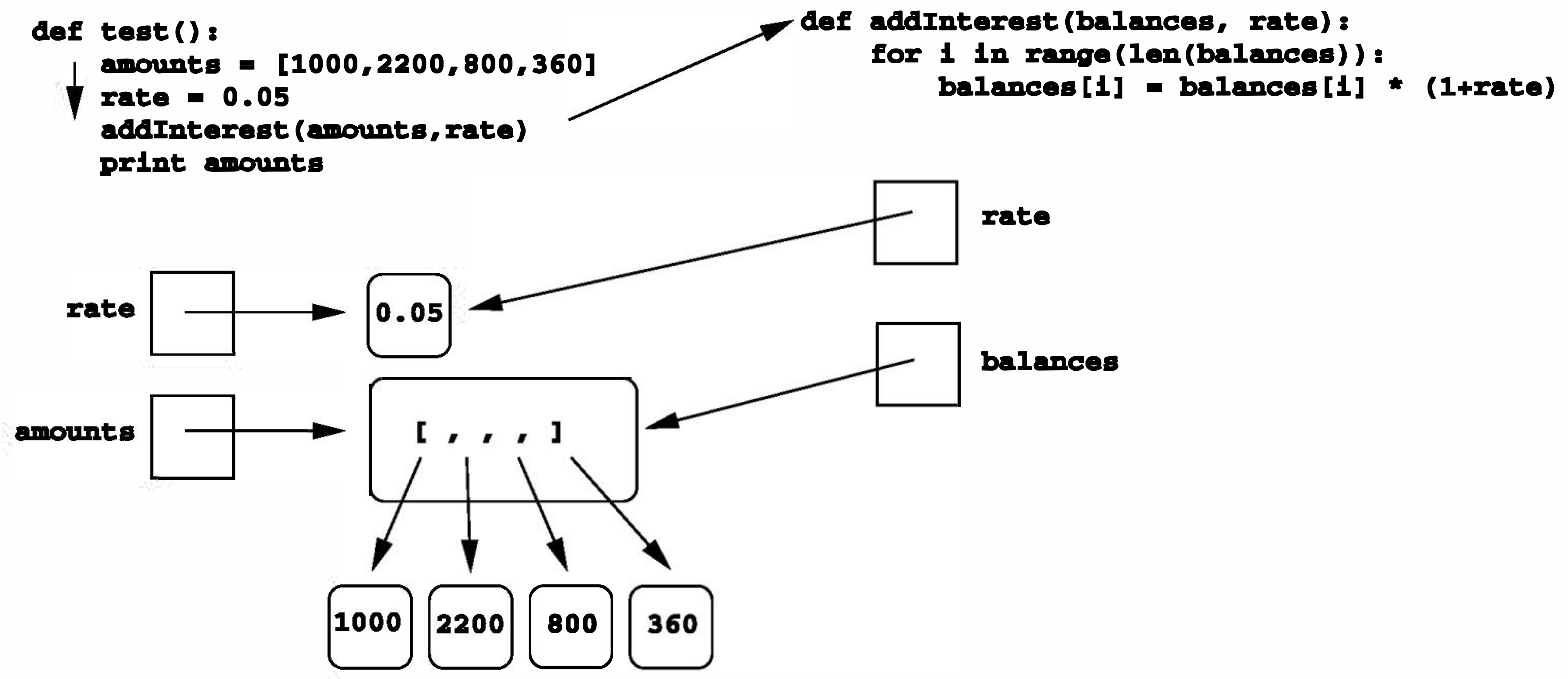
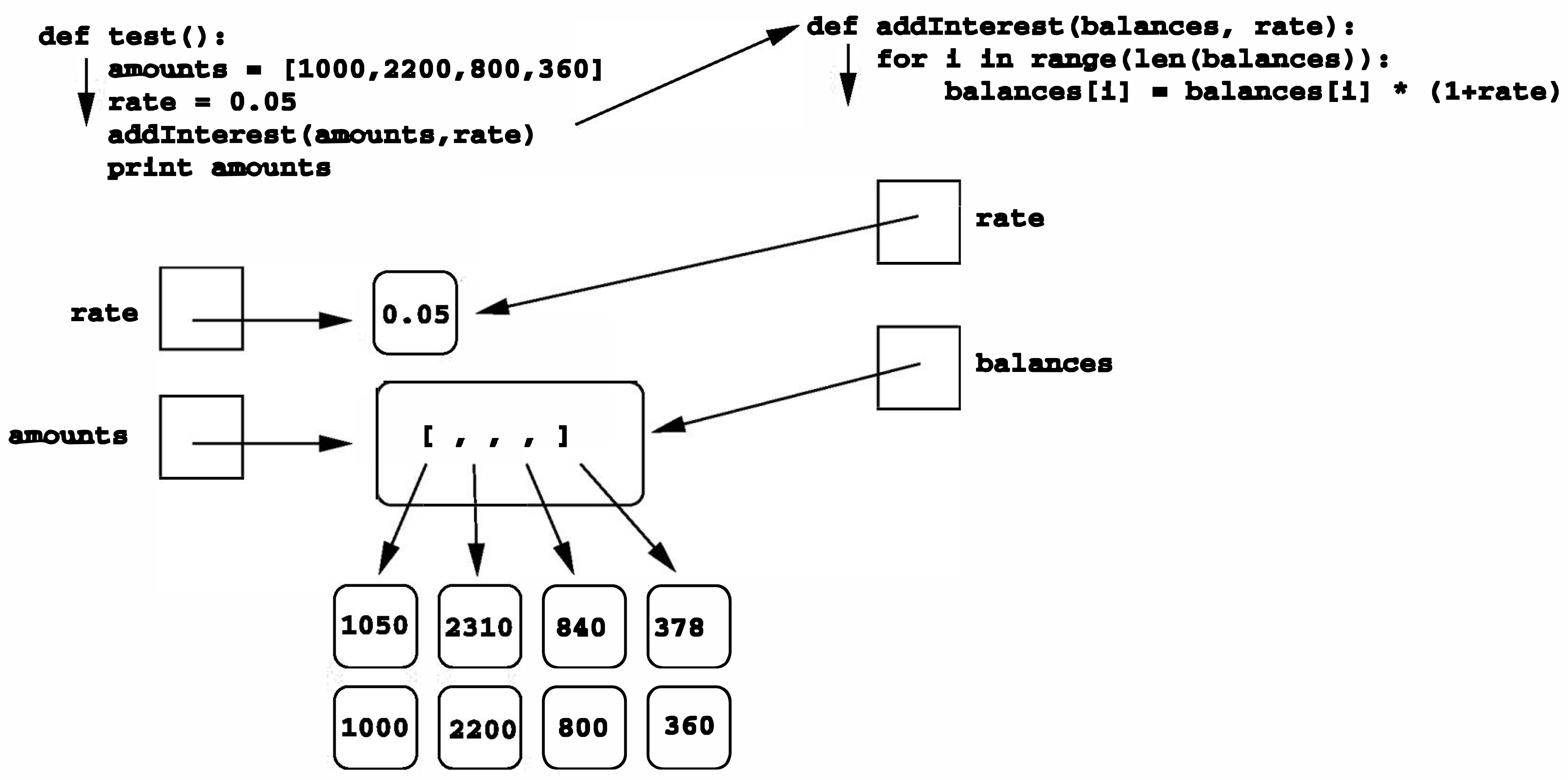

def test():
amounts = [1000,2200,800,360]
rate = 0.05
addInterest(amounts,rate)
print amounts
def addInterest(balances, rate):
for i in range(len(balances)):
balances[i] = balances[i] * (1+rate)
rate
amounts
0.05
[, , ,]
1000
2200
800
360
rate
balances

Figure 6.8: Transfer of list parameter to `addInterest`

Notice that the value of the variable `amounts` is now a list object that itself contains four int values. It is this list object that gets passed to `addInterest` and is therefore also the value of `balances`.

Next, `addInterest` executes. The loop goes through each index in the range $0, 1, \ldots, length - 1$ and updates that item in `balances`. The result is shown in Figure 6.9.



```
def test():
    amounts = [1000,2200,800,360]
    rate = 0.05
    addInterest(amounts,rate)
    print amounts
```

```
def addInterest(balances, rate):
    for i in range(len(balances)):
        balances[i] = balances[i] * (1+rate)
```

Figure 6.9: List modified in `addInterest`

You'll notice in the diagram that I left the old values (1000, 2200, 800, 360) just hanging around. I did this to emphasize that the numbers in the value boxes have not changed. Instead, what has happened is that new values were created, and the assignments into the list caused it to refer to the new values. The old values will actually get cleaned up when Python does garbage collection.

It should be clear now why the list version of the `addInterest` program produces the answer that it does. When `addInterest` terminates, the list stored in `amounts` now contains the new balances, and that is what gets printed. The important point here is that the variable `amounts` *was never changed.* It still refers to the same list that it did before the call to `addInterest`. What has happened is that the state of that list has changed, and this change is visible back in the calling program.

Now you really know everything there is to know about how Python passes parameters to functions. Parameters are always passed by value. However, if the actual parameter is a variable whose value is a mutable object (like a list or graphics object), then changes to the state of the object *will* be visible to the calling program. This situation is another example of the aliasing issue discussed in Chapter 4.

6.7 Functions and Program Structure

So far, we have been discussing functions as a mechanism for reducing code duplication, thus shortening and simplifying our programs. Surprisingly, functions are often used even when doing so actually makes a program longer. A second reason for using functions is to make programs more *modular*.

As the algorithms that you design get more complex, it gets more and more difficult to make sense of programs. Humans are pretty good at keeping track of eight to ten things at a time. When presented with an algorithm that is hundreds of lines long, even the best programmers will throw up their hands in bewilderment.

One way to deal with this complexity is to break an algorithm into smaller subprograms, each of which makes sense on its own. I'll have a lot more to say about this later when we discuss program design in Chapter 9. For now, we'll just take a look at an example. Let's return to the future value problem one more time. Here is the `main` program as we left it:

```
def main():
    # Introduction
    print("This program plots the growth of a 10-year investment.")

    # Get principal and interest rate
    principal = float(input("Enter the initial principal: "))
    apr = float(input("Enter the annualized interest rate: "))

    # Create a graphics window with labels on left edge
    win = GraphWin("Investment Growth Chart", 320, 240)
    win.setBackground("white")
    win.setCoords(-1.75,-200, 11.5, 10400)
    Text(Point(-1, 0), ' 0.0K').draw(win)
    Text(Point(-1, 2500), ' 2.5K').draw(win)
    Text(Point(-1, 5000), ' 5.0K').draw(win)
    Text(Point(-1, 7500), ' 7.5k').draw(win)
    Text(Point(-1, 10000), '10.0K').draw(win)

    # Draw bar for initial principal
    drawBar(win, 0, principal)

    # Draw a bar for each subsequent year
    for year in range(1, 11):
        principal = principal * (1 + apr)
        drawBar(win, year, principal)

    input("Press <Enter> to quit.")
    win.close()

main()
```

Although we have already shortened this algorithm through the use of the drawBar function, it is still long enough to make reading through it awkward. The comments help to explain things, but—not to put too fine a point on it—this function is just too long. One way to make the program more readable is to move some of the details into a separate function. For example, there are eight lines in the middle that simply create the window where the chart will be drawn. We could put these steps into a value-returning function:

```
def createLabeledWindow():
    # Returns a GraphWin with title and labels drawn
    window = GraphWin("Investment Growth Chart", 320, 240)
    window.setBackground("white")
    window.setCoords(-1.75,-200, 11.5, 10400)
    Text(Point(-1, 0), ' 0.0K').draw(window)
    Text(Point(-1, 2500), ' 2.5K').draw(window)
    Text(Point(-1, 5000), ' 5.0K').draw(window)
    Text(Point(-1, 7500), ' 7.5k').draw(window)
    Text(Point(-1, 10000), '10.0K').draw(window)
    return window
```

As its name implies, this function takes care of all the nitty-gritty details of drawing the initial window. It is a self-contained entity that performs this one well-defined task.

Using our new function, the main algorithm seems much simpler:

```
def main():
    print("This program plots the growth of a 10-year investment.")

    principal = input("Enter the initial principal: ")
    apr = input("Enter the annualized interest rate: ")

    win = createLabeledWindow()
    drawBar(win, 0, principal)
    for year in range(1, 11):
        principal = principal * (1 + apr)
        drawBar(win, year, principal)

    input("Press <Enter> to quit.")
    win.close()
```

Notice that I have removed the comments; the intent of the algorithm is now clear. With suitably named functions, the code has become nearly self-documenting.

Here is the final version of our future value program:

```
# futval_graph4.py

from graphics import *

def createLabeledWindow():
    window = GraphWin("Investment Growth Chart", 320, 240)
    window.setBackground("white")
    window.setCoords(-1.75,-200, 11.5, 10400)
    Text(Point(-1, 0), ' 0.0K').draw(window)
    Text(Point(-1, 2500), ' 2.5K').draw(window)
    Text(Point(-1, 5000), ' 5.0K').draw(window)
    Text(Point(-1, 7500), ' 7.5k').draw(window)
    Text(Point(-1, 10000), '10.0K').draw(window)
    return window

def drawBar(window, year, height):
    bar = Rectangle(Point(year, 0), Point(year+1, height))
    bar.setFill("green")
    bar.setWidth(2)
    bar.draw(window)
```

```
def main():
    print("This program plots the growth of a 10 year investment.")

    principal = float(input("Enter the initial principal: "))
    apr = float(input("Enter the annualized interest rate: "))

    win = createLabeledWindow()
    drawBar(win, 0, principal)
    for year in range(1, 11):
        principal = principal * (1 + apr)
        drawBar(win, year, principal)

    input("Press <Enter> to quit.")
    win.close()

main()
```

Although this version is longer than the previous version, experienced programmers would find it much easier to understand. As you get used to reading and writing functions, you too will learn to appreciate the elegance of more modular code.

6.8 Chapter Summary

- A function is a kind of subprogram. Programmers use functions to reduce code duplication and to help structure or modularize programs. Once a function is defined, it may be called multiple times from many different places in a program. Parameters allow functions to have changeable parts. The parameters appearing in the function definition are called formal parameters, and the expressions appearing in a function call are known as actual parameters.
- A call to a function initiates a four-step process:
  1. The calling program is suspended.
  2. The values of actual parameters are assigned to the formal parameters.
  3. The body of the function is executed.

   4. Control returns immediately following the function call in the calling program. The value returned by the function is used as the expression result.

- The scope of a variable is the area of the program where it may be referenced. Formal parameters and other variables inside function definitions are local to the function. Local variables are distinct from variables of the same name that may be used elsewhere in the program.
- Functions can communicate information back to the caller through return values. In Python, functions may return multiple values. Value-returning functions should generally be called from inside an expression. Functions that don't explicitly return a value return the special object `None`.
- Python passes parameters by value. If the value being passed is a mutable object, then changes made to the object may be visible to the caller.

6.9 Exercises

Review Questions

True/False

1. Programmers rarely define their own functions.
2. A function may only be called at one place in a program.
3. Information can be passed into a function through parameters.
4. Every Python function returns some value.
5. In Python, some parameters are passed by reference.
6. In Python, a function can return only one value.
7. Python functions can never modify a parameter.
8. One reason to use functions is to reduce code duplication.
9. Variables defined in a function are local to that function.
10. It's a bad idea to define new functions if it makes a program longer.

Multiple Choice

1. The part of a program that uses a function is called the
   a) user b) caller c) callee d) statement

2. A Python function definition begins with
   a) `def` b) `define` c) `function` d) `defun`

3. A function can send output back to the program with a(n)
   a) `return` b) `print` c) assignment d) SASE

4. Formal and actual parameters are matched up by
   a) name b) position c) ID d) interests

5. Which of the following is *not* a step in the function-calling process?
   a) The calling program suspends.
   b) The formal parameters are assigned the value of the actual parameters.
   c) The body of the function executes.
   d) Control returns to the point just before the function was called.

6. In Python, actual parameters are passed to functions
   a) by value b) by reference c) at random d) by networking

7. Which of the following is *not* a reason to use functions?
   a) to reduce code duplication
   b) to make a program more modular
   c) to make a program more self-documenting
   d) to demonstrate intellectual superiority

8. If a function returns a value, it should generally be called from
   a) an expression b) a different program
   c) `main` d) a cell phone

9. A function with no `return` statement returns
   a) nothing b) its parameters c) its variables d) `None`

10. A function can modify the value of an actual parameter only if it's
    a) mutable b) a list c) passed by reference d) a variable

Discussion

1. In your own words, describe the two motivations for defining functions in your programs.

2. We have been thinking about computer programs as sequences of instructions where the computer methodically executes one instruction and then moves on to the next one. Do programs that contain functions fit this model? Explain your answer.

3. Parameters are an important concept in defining functions.

   a) What is the purpose of parameters?

   b) What is the difference between a formal parameter and an actual parameter?

   c) In what ways are parameters similar to and different from ordinary variables?

4. Functions can be thought of as miniature (sub)programs inside other programs. Like any other program, we can think of functions as having input and output to communicate with the main program.

   a) How does a program provide "input" to one of its functions?

   b) How does a function provide "output" to the program?

5. Consider this very simple function:

```
def cube(x):
    answer = x * x * x
    return answer
```

   a) What does this function do?

   b) Show how a program could use this function to print the value of y^3, assuming y is a variable.

   c) Here is a fragment of a program that uses this function:

```
answer = 4
result = cube(3)
print(answer, result)
```

   The output from this fragment is `4 27`. Explain why the output is not `27 27`, even though `cube` seems to change the value of `answer` to 27.

Programming Exercises

1. Write a program to print the lyrics of the song "Old MacDonald." Your program should print the lyrics for five different animals, similar to the example verse below.

   Old MacDonald had a farm, Ee-igh, Ee-igh, Oh!
   And on that farm he had a cow, Ee-igh, Ee-igh, Oh!
   With a moo, moo here and a moo, moo there.
   Here a moo, there a moo, everywhere a moo, moo.
   Old MacDonald had a farm, Ee-igh, Ee-igh, Oh!

2. Write a program to print the lyrics for ten verses of "The Ants Go Marching." A couple of sample verses are given below. You may choose your own activity for the "little one" in each verse, but be sure to choose something that makes the rhyme work (or almost work).

   The ants go marching one by one, hurrah! hurrah!
   The ants go marching one by one, hurrah! hurrah!
   The ants go marching one by one,
   The little one stops to suck his thumb,
   And they all go marching down...
   In the ground...
   To get out....
   Of the rain.
   Boom! Boom! Boom!

   The ants go marching two by two, hurrah! hurrah!
   The ants go marching two by two, hurrah! hurrah!
   The ants go marching two by two,
   The little one stops to tie his shoe,
   And they all go marching down...
   In the ground...
   To get out...
   Of the rain.
   Boom! Boom! Boom!

3. Write definitions for these functions:

   `sphereArea(radius)` Returns the surface area of a sphere having the given radius.

`sphereVolume(radius)` Returns the volume of a sphere having the given radius.

Use your functions to solve Programming Exercise 1 from Chapter 3.

4. Write definitions for the following two functions:

   `sumN(n)` returns the sum of the first `n` natural numbers.

   `sumNCubes(n)` returns the sum of the cubes of the first `n` natural numbers.

   Then use these functions in a program that prompts a user for an n and prints out the sum of the first n natural numbers and the sum of the cubes of the first n natural numbers.

5. Redo Programming Exercise 2 from Chapter 3. Use two functions—one to compute the area of a pizza, and one to compute cost per square inch.

6. Write a function that computes the area of a triangle given the length of its three sides as parameters (see Programming Exercise 9 from Chapter 3). Use your function to augment `triangle2.py` from this chapter so that it also displays the area of the triangle.

7. Write a function to compute the nth Fibonacci number. Use your function to solve Programming Exercise 16 from Chapter 3.

8. Solve Programming Exercise 17 from Chapter 3 using a function `nextGuess(guess, x)` that returns the next guess.

9. Do Programming Exercise 3 from Chapter 5 using a function `grade(score)` that returns the letter grade for a score.

10. Do Programming Exercise 4 from Chapter 5 using a function `acronym(phrase)` that returns an acronym for a phrase supplied as a string.

11. Write and test a function to meet this specification.

    `squareEach(nums)` `nums` is a list of numbers. Modifies the list by squaring each entry.

12. Write and test a function to meet this specification.

    `sumList(nums)` `nums` is a list of numbers. Returns the sum of the numbers in the list.

13. Write and test a function to meet this specification.

    `toNumbers(strList)` `strList` is a list of strings, each of which represents a number. Modifies each entry in the list by converting it to a number.

14. Use the functions from the previous three problems to implement a program that computes the sum of the squares of numbers read from a file. Your program should prompt for a file name and print out the sum of the squares of the values in the file. *Hint*: Use `readlines()`

15. Write and test a function to meet this specification.

    `drawFace(center, size, win)` `center` is a `Point`, `size` is an int, and `win` is a `GraphWin`. Draws a simple face of the given size in `win`.

    Your function can draw a simple smiley (or grim) face. Demonstrate the function by writing a program that draws several faces of varying size in a single window.

16. Use your `drawFace` function from the previous exercise to write a photo anonymizer. This program allows a user to load an image file (such as a PPM or GIF) and to draw cartoon faces over the top of existing faces in the photo. The user first inputs the name of the file containing the image. The image is displayed and the user is asked how many faces are to be blocked. The program then enters a loop for the user to click on two points for each face: the center and somewhere on the edge of the face (to determine the size of the face). The program should then draw a face in that location using the `drawFace` function.

    Hints: Section 4.8.4 describes the image-manipulation methods in the graphics library. Display the image centered in a `GraphWin` that is the same width and height as the image, and draw the graphics into this window. You can use a screen capture utility to save the resulting images.

17. Write a function to meet this specification.

    `moveTo(shape, newCenter)` `shape` is a graphics object that supports the `getCenter` method and `newCenter` is a `Point`. Moves `shape` so that `newCenter` is its center.

    Use your function to write a program that draws a circle and then allows the user to click the window 10 times. Each time the user clicks, the circle is moved where the user clicked.

Chapter 7 Decision Structures

Objectives

- To understand the simple decision programming pattern and its implementation using a Python `if` statement.
- To understand the two-way decision programming pattern and its implementation using a Python `if-else` statement.
- To understand the multi-way decision programming pattern and its implementation using a Python `if-elif-else` statement.
- To understand the idea of exception handling and be able to write simple exception-handling code that catches standard Python run-time errors.
- To understand the concept of Boolean expressions and the `bool` data type.
- To be able to read, write, and implement algorithms that employ decision structures, including those that employ sequences of decisions and nested decision structures.

7.1 Simple Decisions

So far, we have mostly viewed computer programs as sequences of instructions that are followed one after the other. Sequencing is a fundamental concept of programming, but alone it is not sufficient to solve every problem. Often it is necessary to alter the sequential flow of a program to suit the needs of a particular situation. This is done with special statements known as *control structures*. In this chapter, we'll take a look at *decision structures*, which are statements that allow a program to execute different sequences of instructions for different cases, effectively allowing the program to "choose" an appropriate course of action.

7.1.1 Example: Temperature Warnings

Let's start by getting the computer to make a simple decision. For an easy example, we'll return to the Celsius to Fahrenheit temperature conversion program from Chapter 2. Remember, this was written by Susan Computewell to help her figure out how to dress each morning in Europe. Here is the program as we left it:

```
# convert.py
#     A program to convert Celsius temps to Fahrenheit
# by: Susan Computewell

def main():
    celsius = float(input("What is the Celsius temperature? "))
    fahrenheit = 9/5 * celsius + 32
    print("The temperature is", fahrenheit, "degrees fahrenheit.")

main()
```

This is a fine program as far as it goes, but we want to enhance it. Susan Computewell is not a morning person, and even though she has a program to convert the temperatures, sometimes she does not pay very close attention to the results. Our enhancement to the program will ensure that when the temperatures are extreme, the program prints out a suitable warning so that Susan takes notice.

The first step is to fully specify the enhancement. An extreme temperature is either quite hot or quite cold. Let's say that any temperature over 90 degrees Fahrenheit deserves a heat warning, and a temperature under 30 degrees warrants a cold warning. With this specification in mind, we can design an extended algorithm:

```
Input the temperature in degrees Celsius (call it celsius)
Calculate fahrenheit as 9/5 celsius + 32
Output fahrenheit
```

```
if fahrenheit > 90
   print a heat warning
if fahrenheit < 30
   print a cold warning
```

This new design has two simple *decisions* at the end. The indentation indicates that a step should be performed only if the condition listed in the previous line is met. The idea here is that the decision introduces an alternative flow of control through the program. The exact set of steps taken by the algorithm will depend on the value of `fahrenheit`.

Figure 7.1 is a flowchart showing the possible paths that can be taken through the algorithm. The diamond boxes show conditional decisions. If the condition is false, control passes to the next statement in the sequence (the one below). If the condition holds, however, control transfers to the instructions in the box to the right. Once these instructions are done, control then passes to the next statement.

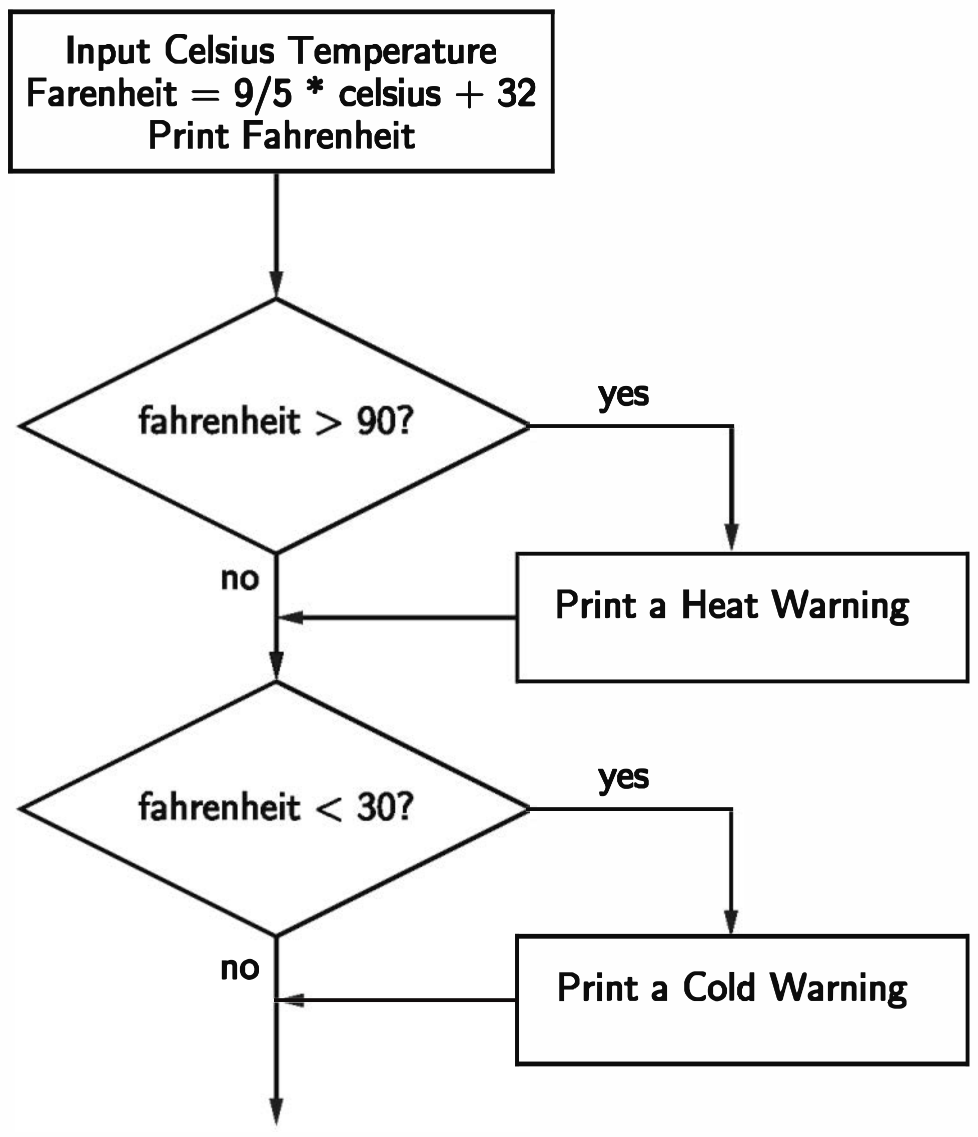


Figure 7.1: Flowchart of temperature conversion program with warnings

Here is how the new design translates into Python code:

```
# convert2.py
#      A program to convert Celsius temps to Fahrenheit.
#      This version issues heat and cold warnings.

def main():
    celsius = float(input("What is the Celsius temperature? "))
    fahrenheit = 9/5 * celsius + 32
    print("The temperature is", fahrenheit, "degrees Fahrenheit.")

    # Print warnings for extreme temps
    if fahrenheit > 90:
        print("It's really hot out there. Be careful!")
    if fahrenheit < 30:
        print("Brrrrr. Be sure to dress warmly!")

main()
```

You can see that the Python `if` statement is used to implement the decision.

The form of the `if` is very similar to the pseudocode in the algorithm.

```
if <condition>:
    <body>
```

The `body` is just a sequence of one or more statements indented under the `if` heading. In `convert2.py` there are two `if` statements, both of which have a single statement in the body.

The semantics of the `if` should be clear from the example above. First, the condition in the heading is evaluated. If the condition is true, the sequence of statements in the body is executed, and then control passes to the next statement in the program. If the condition is false, the statements in the body are skipped. Figure 7.2 shows the semantics of the `if` as a flowchart. Notice that the body of the `if` either executes or not depending on the condition. In either case, control then passes to the next statement after the `if`. This is a *one-way* or *simple* decision.

7.1.2 Forming Simple Conditions

One point that has not yet been discussed is exactly what a condition looks like. For the time being, our programs will use simple conditions that compare the values of two expressions: `<expr> <relop> <expr>`. Here `<relop>` is short for

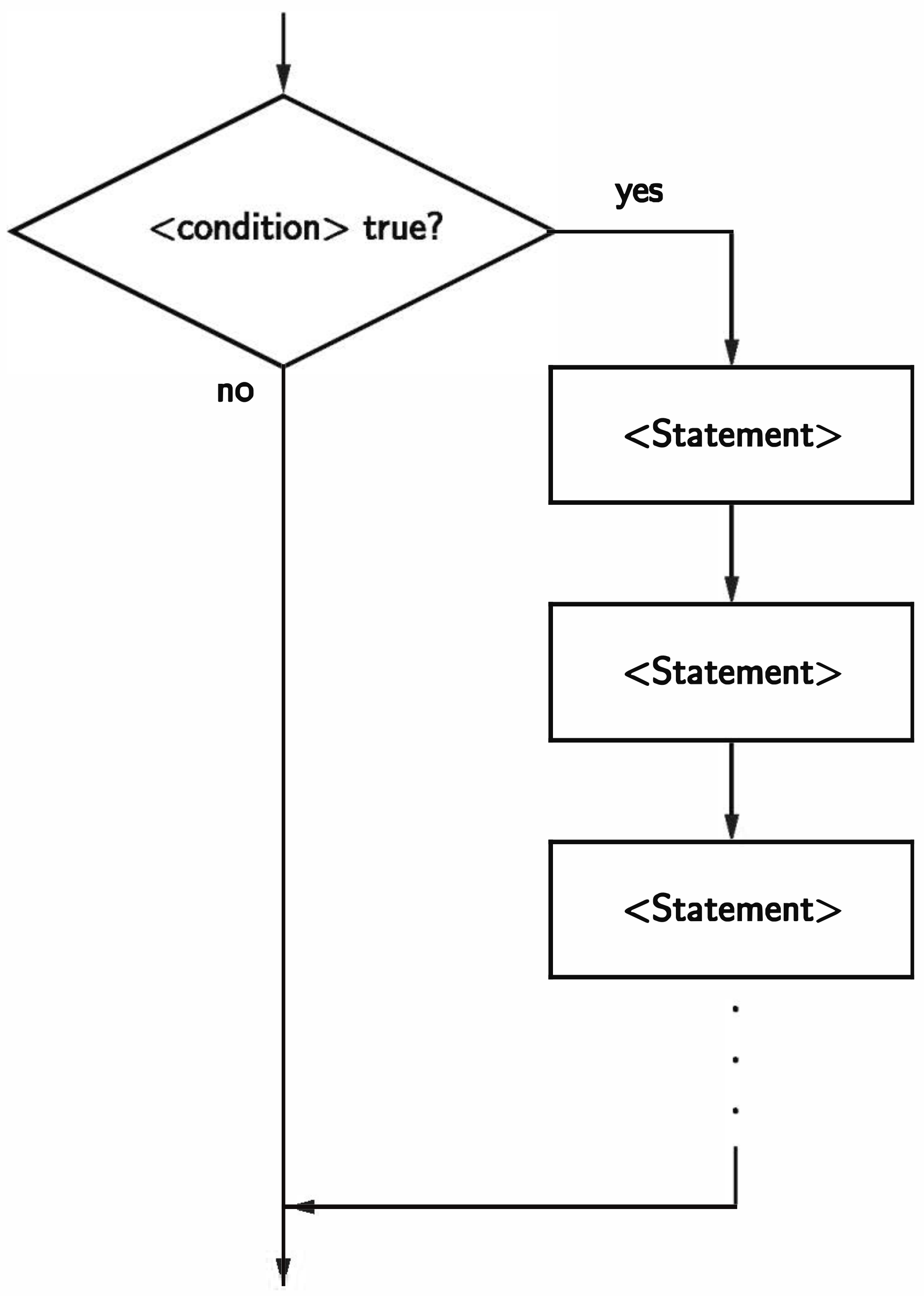


Figure 7.2: Control flow of simple `if` statement

relational operator. That's just a fancy name for the mathematical concepts like "less than" or "equal to." There are six relational operators in Python, shown in the following table:

Python	mathematics	meaning
<	$<$	less than
<=	$\leq$	less than or equal to
==	$=$	equal to
>=	$\geq$	greater than or equal to
>	$>$	greater than
!=	$\neq$	not equal to

Notice especially the use of == for equality. Since Python uses the = sign to indicate an assignment statement, a different symbol is required for the concept of equality. A common mistake in Python programs is using = in conditions, where a == is required.

Conditions may compare either numbers or strings. When comparing strings, the ordering is *lexicographic*. Basically, this means that strings are put in alphabetic order according to the underlying Unicode values. So all uppercase Latin letters come before lowercase equivalents (e.g., "Bbbb" comes before "aaaa," since "B" precedes "a").

I should mention that conditions are actually a type of expression, called a *Boolean* expression, after George Boole, a 19th century English mathematician. When a Boolean expression is evaluated, it produces a value of either *true* (the condition holds) or *false* (it does not hold). Some languages such as C++ and older versions of Python just use the ints 1 and 0 to represent these values. Other languages like Java and modern Python have a dedicated data type for Boolean expressions.

In Python, Boolean expressions are of type `bool` and the Boolean values true and false are represented by the literals `True` and `False`. Here are a few interactive examples:

```
>>> 3 < 4
True
>>> 3 * 4 < 3 + 4
False
>>> "hello" == "hello"
True
>>> "hello" < "hello"
False
>>> "Hello" < "hello"
True
```

7.1.3 Example: Conditional Program Execution

Back in Chapter 1, I mentioned that there are several different ways of running Python programs. Some Python module files are designed to be run directly. These are usually referred to as "programs" or "scripts." Other Python modules are designed primarily to be imported and used by other programs; these are often called "libraries." Sometimes we want to create a sort of hybrid module that can be used both as a stand-alone program and as a library that can be imported by other programs.

So far, most of our programs have had a line at the bottom to invoke the `main` function.

```
main()
```

As you know, this is what actually starts a program running. These programs are suitable for running directly. In a windowing environment, you might run a file by (double-)clicking its icon. Or you might type a command like `python <myfile>.py`.

Since Python evaluates the lines of a module during the import process, our current programs also run when they are imported into either an interactive Python session or into another Python program. Generally, it is nicer not to have modules run as they are imported. When testing a program interactively, the usual approach is to first import the module and then call its `main` (or some other function) each time we want to run it.

In a program designed to be *either* imported (without running) *or* run directly, the call to `main` at the bottom must be made conditional. A simple decision should do the trick:

```
if <condition>:
    main()
```

We just need to figure out a suitable condition.

Whenever a module is imported, Python creates a special variable, `__name__`, inside that module and assigns it a string representing the module's name. Here is an example interaction showing what happens with the `math` library:

```
>>> import math
>>> math.__name__
'math'
```

You can see that, when imported, the `__name__` variable inside the `math` module is assigned the string `'math'`.

However, when Python code is being run directly (not imported), Python sets the value of `__name__` to be `'__main__'`. To see this in action, you just need to start a Python shell and look at the value.

```
>>> __name__
'__main__'
```

So if a module is imported, the code inside that module will see a variable called `__name__` whose value is the name of the module. When a file is run directly, the code will see that `__name__` has the value `'__main__'`. A module can determine how it is being used by inspecting this variable.

Putting the pieces together, we can change the final lines of our programs to look like this:

```
if __name__ == '__main__':
    main()
```

This guarantees that `main` will automatically run when the program is invoked directly, but it will not run if the module is imported. You will see a line of code similar to this at the bottom of virtually every Python program.

7.2 Two-Way Decisions

Now that we have a way to selectively execute certain statements in a program using decisions, it's time to go back and spruce up the quadratic equation solver from Chapter 3. Here is the program as we left it:

```
# quadratic.py
#    A program that computes the real roots of a quadratic equation.
#    Note: This program crashes if the equation has no real roots.

import math

def main():
    print("This program finds the real solutions to a quadratic\n")

    a = float(input("Enter coefficient a: "))
    b = float(input("Enter coefficient b: "))
    c = float(input("Enter coefficient c: "))

    discRoot = math.sqrt(b * b - 4 * a * c)
    root1 = (-b + discRoot) / (2 * a)
    root2 = (-b - discRoot) / (2 * a)

    print("\nThe solutions are:", root1, root2 )

main()
```

As noted in the comments, this program crashes when it is given coefficients of a quadratic equation that has no real roots. The problem with this code is that when $b^2 - 4ac$ is less than 0, the program attempts to take the square root of a negative number. Since negative numbers do not have real roots, the `math` library reports an error. Here's an example:

```
>>> main()
This program finds the real solutions to a quadratic

Enter coefficient a: 1
Enter coefficient b: 2
Enter coefficient c: 3
Traceback (most recent call last):
  File "quadratic.py", line 23, in <module>
    main()
  File "quadratic.py", line 16, in main
    discRoot = math.sqrt(b * b - 4 * a * c)
ValueError: math domain error
```

We can use a decision to check for this situation and make sure that the program can't crash. Here's a first attempt:

```
# quadratic2.py
import math

def main():
    print("This program finds the real solutions to a quadratic\n")
    a = float(input("Enter coefficient a: "))
    b = float(input("Enter coefficient b: "))
    c = float(input("Enter coefficient c: "))

    discrim = b * b - 4 * a * c
    if discrim >= 0:
        discRoot = math.sqrt(discrim)
        root1 = (-b + discRoot) / (2 * a)
        root2 = (-b - discRoot) / (2 * a)
        print("\nThe solutions are:", root1, root2 )

main()
```

This version first computes the value of the discriminant ($b^2 - 4ac$) and then checks to make sure it is not negative. Only then does the program proceed to take the square root and calculate the solutions. This program will never attempt to call `math.sqrt` when `discrim` is negative.

Unfortunately, this updated version is not really a complete solution. Do you see what happens when the equation has no real roots? According to the semantics for a simple `if`, when `b*b - 4*a*c` is less than zero, the program will simply skip the calculations and go to the next statement. Since there is no next statement, the program just quits. Here's an example interactive session:

```
>>> main()
This program finds the real solutions to a quadratic

Enter coefficient a: 1
Enter coefficient b: 2
Enter coefficient c: 3
>>>
```

This is almost worse than the previous version, because it does not give users any indication of what went wrong; it just leaves them hanging. A better program would print a message telling users that their particular equation has no real solutions. We could accomplish this by adding another simple decision at the end of the program.

```
if discrim < 0:
    print("The equation has no real roots!")
```

This will certainly solve our problem, but this solution just doesn't feel right. We have programmed a sequence of two decisions, but the two outcomes are mutually exclusive. If `discrim >= 0` is true then `discrim < 0` must be false and vice versa. We have two conditions in the program, but there is really only one decision to make. Based on the value of `discrim` the program should *either* print that there are no real roots *or* it should calculate and display the roots. This is an example of a two-way decision. Figure 7.3 illustrates the situation.

In Python, a two-way decision can be implemented by attaching an `else` clause onto an `if` clause. The result is called an `if-else` statement.

```
if <condition>:
    <statements>
else:
    <statements>
```

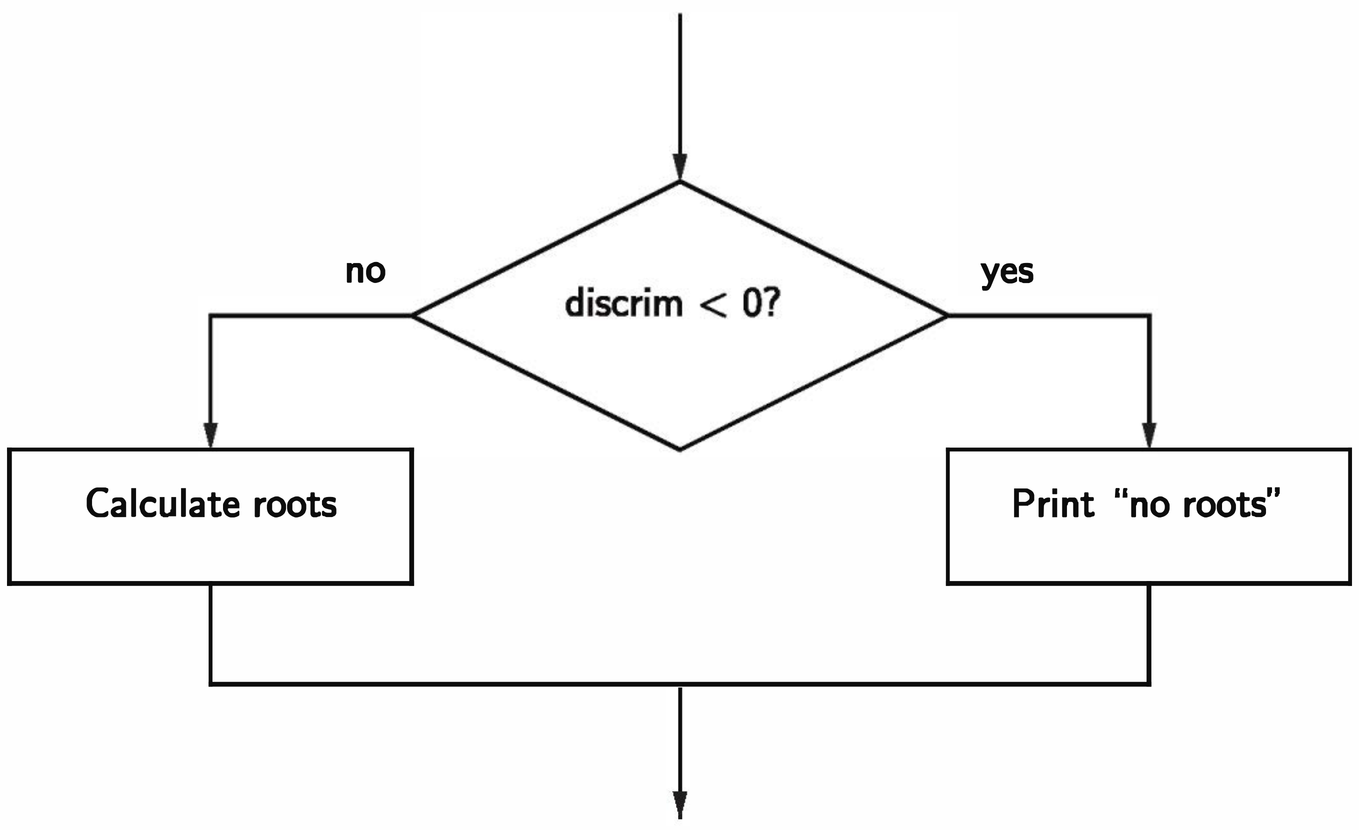


Figure 7.3: Quadratic solver as a two-way decision

When the Python interpreter encounters this structure, it will first evaluate the condition. If the condition is true, the statements under the if are executed. If the condition is false, the statements under the else are executed. In either case, control then passes to the statement following the if-else.

Using a two-way decision in the quadratic solver yields a more elegant solution:

```
# quadratic3.py
import math

def main():
    print("This program finds the real solutions to a quadratic\n")

    a = float(input("Enter coefficient a: "))
    b = float(input("Enter coefficient b: "))
    c = float(input("Enter coefficient c: "))

    discrim = b * b - 4 * a * c
    if discrim < 0:
        print("\nThe equation has no real roots!")
    else:
        discRoot = math.sqrt(b * b - 4 * a * c)
        root1 = (-b + discRoot) / (2 * a)
        root2 = (-b - discRoot) / (2 * a)
        print("\nThe solutions are:", root1, root2)
main()
```

This program fills the bill nicely. Here is a sample session that runs the new program twice:

```
>>> main()
This program finds the real solutions to a quadratic

Enter coefficient a: 1
Enter coefficient b: 2
Enter coefficient c: 3

The equation has no real roots!
>>> main()
This program finds the real solutions to a quadratic

Enter coefficient a: 2
Enter coefficient b: 4
Enter coefficient c: 1

The solutions are: -0.2928932188134524 -1.7071067811865475
>>>
```

7.3 Multi-Way Decisions

The newest version of the quadratic solver is certainly a big improvement, but it still has some quirks. Here is another example run:

```
>>> main()
This program finds the real solutions to a quadratic

Enter coefficient a: 1
Enter coefficient b: 2
Enter coefficient c: 1

The solutions are: -1.0 -1.0
```

This is technically correct; the given coefficients produce an equation that has a double root at -1. However, the output might be confusing to some users. It looks like the program has mistakenly printed the same number twice. Perhaps the program should be a bit more informative to avoid confusion.

The double-root situation occurs when `discrim` is exactly 0. In this case, `discRoot` is also 0, and both roots have the value $\frac{-b}{2a}$. If we want to catch this special case, our program actually needs a three-way decision. Here's a quick sketch of the design:

```
...
Check the value of discrim
    when < 0: handle the case of no roots
    when = 0: handle the case of a double root
    when > 0: handle the case of two distinct roots.
```

One way to code this algorithm is to use two `if-else` statements. The body of an `if` or `else` clause can contain any legal Python statements, including other `if` or `if-else` statements. Putting one compound statement inside another is called *nesting*. Here's a fragment of code that uses nesting to achieve a three-way decision:

```
if discrim < 0:
    print("Equation has no real roots")
else:
    if discrim == 0:
        root = -b / (2 * a)
        print("There is a double root at", root)
    else:
        # Do stuff for two roots
```

If you trace through this code carefully, you will see that there are exactly three possible paths. The sequencing is determined by the value of `discrim`. A flowchart of this solution is shown in Figure 7.4. You can see that the top-level structure is just an `if-else`. (Treat the dashed box as one big statement.) The dashed box contains the second `if-else` nested comfortably inside the `else` part of the top-level decision.

Once again, we have a working solution, but the implementation doesn't feel quite right. We have finessed a three-way decision by using two two-way decisions. The resulting code does not reflect the true three-fold decision of the original problem. Imagine if we needed to make a five-way decision like this.

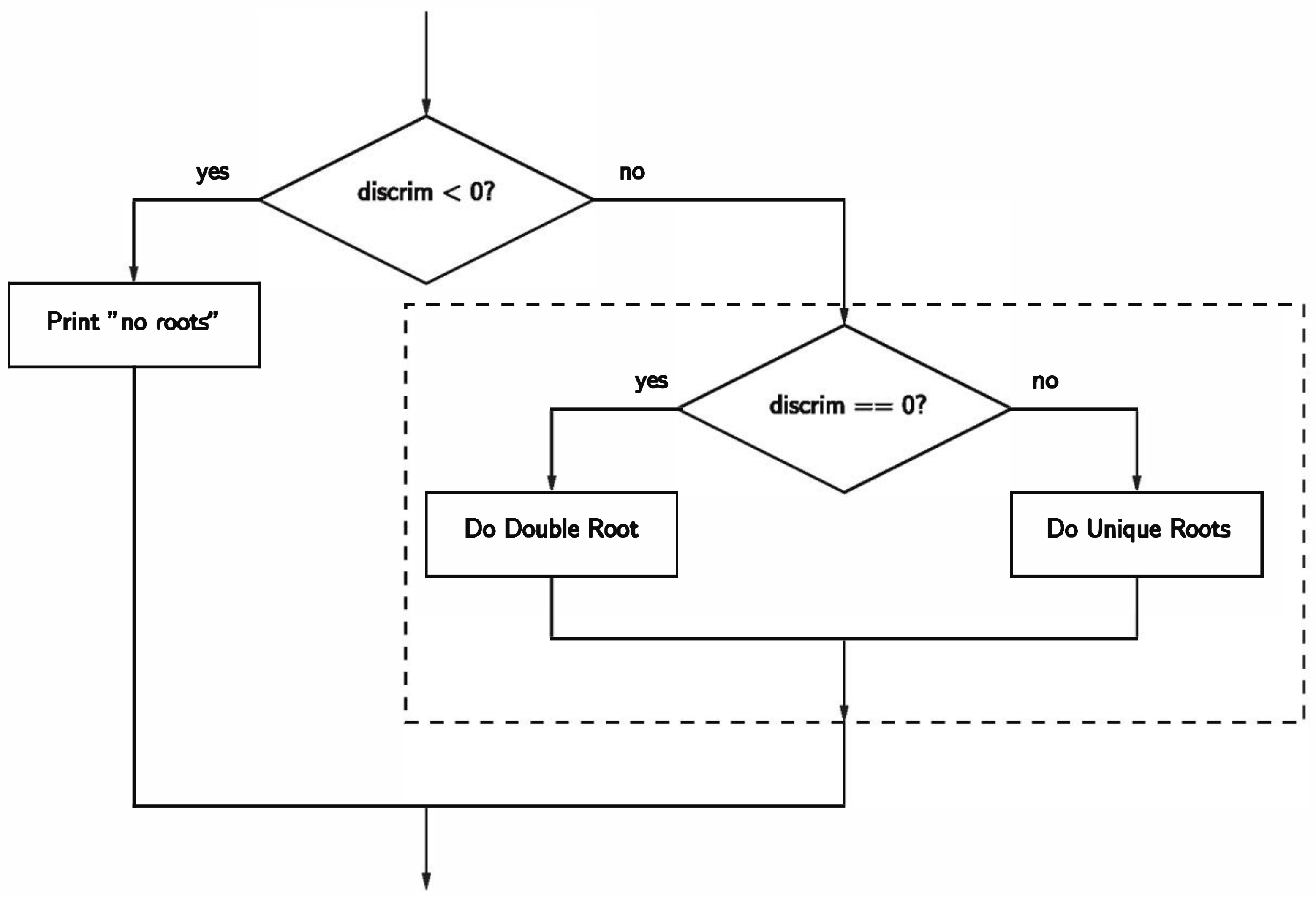


Figure 7.4: Three-way decision for quadratic solver using nested `if-else`

The `if-else` structures would nest four levels deep, and the Python code would march off the right-hand edge of the page.

There is another way to write multi-way decisions in Python that preserves the semantics of the nested structures but gives it a more appealing look. The idea is to combine an `else` followed immediately by an `if` into a single clause called an `elif` (pronounced "ell-if").

```
if <condition1>:
    <case1 statements>
elif <condition2>:
    <case2 statements>
elif <condition3>:
    <case3 statements>
...
else:
    <default statements>
```

This form is used to set off any number of mutually exclusive code blocks. Python will evaluate each condition in turn looking for the first one that is true. If a true condition is found, the statements indented under that condition are executed, and control passes to the next statement after the entire `if-elif-else`.

If none of the conditions are true, the statements under the `else` are performed. The `else` clause is optional; if omitted, it is possible that no indented statement block will be executed.

Using an `if-elif-else` to show the three-way decision in our quadratic solver yields a nicely finished program:

```
# quadratic4.py
import math

def main():
    print("This program finds the real solutions to a quadratic\n")

    a = float(input("Enter coefficient a: "))
    b = float(input("Enter coefficient b: "))
    c = float(input("Enter coefficient c: "))

    discrim = b * b - 4 * a * c
    if discrim < 0:
        print("\nThe equation has no real roots!")
    elif discrim == 0:
        root = -b / (2 * a)
        print("\nThere is a double root at", root)
    else:
        discRoot = math.sqrt(b * b - 4 * a * c)
        root1 = (-b + discRoot) / (2 * a)
        root2 = (-b - discRoot) / (2 * a)
        print("\nThe solutions are:", root1, root2 )
main()
```

7.4 Exception Handling

Our quadratic program uses decision structures to avoid taking the square root of a negative number and generating an error at runtime. This is a common pattern in many programs: using decisions to protect against rare but possible errors.

In the case of the quadratic solver, we checked the data *before* the call to the `sqrt` function. Sometimes functions themselves check for possible errors and return a special value to indicate that the operation was unsuccessful. For example, a different square root operation might return a negative number (say, -1) to indicate an error. Since the square root function should always return the non-negative root, this value could be used to signal that an error has occurred. The program would check the result of the operation with a decision:

```
discRt = otherSqrt(b*b - 4*a*c)
if discRt < 0:
    print("No real roots.")
else:
    ...
```

Sometimes programs become so peppered with decisions to check for special cases that the main algorithm for handling the run-of-the-mill cases seems completely lost. Programming language designers have come up with mechanisms for *exception handling* that help to solve this design problem. The idea of an exception-handling mechanism is that the programmer can write code that catches and deals with errors that arise when the program is running. Rather than explicitly checking that each step in the algorithm was successful, a program with exception handling can in essence say, "Do these steps, and if any problem crops up, handle it this way."

We're not going to discuss all the details of the Python exception-handling mechanism here, but I do want to give you a concrete example so you can see how exception handling works and understand programs that use it. In Python, exception handling is done with a special control structure that is similar to a decision. Let's start with a specific example and then take a look at the general approach.

Here is a version of the quadratic program that uses Python's exception mechanism to catch potential errors in the `math.sqrt` function:

```
# quadratic5.py
import math

def main():
    print("This program finds the real solutions to a quadratic\n")

    try:
        a = float(input("Enter coefficient a: "))
        b = float(input("Enter coefficient b: "))
        c = float(input("Enter coefficient c: "))
        discRoot = math.sqrt(b * b - 4 * a * c)
        root1 = (-b + discRoot) / (2 * a)
        root2 = (-b - discRoot) / (2 * a)
        print("\nThe solutions are:", root1, root2)
    except ValueError:
        print("\nNo real roots")
main()
```

Notice that this is basically the very first version of the quadratic program with the addition of a `try...except` around the heart of the program. A `try` statement has the general form:

```
try:
    <body>
except <ErrorType>:
    <handler>
```

When Python encounters a `try` statement, it attempts to execute the statements inside the body. If these statements execute without error, control then passes to the next statement after the `try...except`. If an error occurs somewhere in the body, Python looks for an `except` clause with a matching error type. If a suitable `except` is found, the handler code is executed.

The original program *without the exception handling* produced the following error:

```
Traceback (most recent call last):
  File "quadratic.py", line 23, in <module>
    main()
  File "quadratic.py", line 16, in main
    discRoot = math.sqrt(b * b - 4 * a * c)
ValueError: math domain error
```

The last line of this error message indicates the type of error that was generated, namely a `ValueError`. The updated version of the program provides an `except` clause to catch the `ValueError`. Here's how it looks in action:

```
This program finds the real solutions to a quadratic

Enter coefficient a: 1
Enter coefficient b: 2
Enter coefficient c: 3

No real roots
```

Instead of crashing, the exception handler catches the error and prints a message indicating that the equation does not have real roots.

Interestingly, our new program also catches errors caused by the user typing invalid input values. Let's run the program again, and this time type "x" as the first input. Here's how it looks:

```
This program finds the real solutions to a quadratic

Enter coefficient a: x

No real roots
```

Do you see what happened here? Python raised a `ValueError` executing `float("x")` because `"x"` is not convertible to a float. This caused the program to exit the `try` and jump to the `except` clause for that error. Of course, the final message here looks a bit strange. Here's one last version of the program that checks to see what sort of error occurred:

```
# quadratic6.py
import math

def main():
    print("This program finds the real solutions to a quadratic\n")
    try:
        a = float(input("Enter coefficient a: "))
        b = float(input("Enter coefficient b: "))
        c = float(input("Enter coefficient c: "))
        discRoot = math.sqrt(b * b - 4 * a * c)
        root1 = (-b + discRoot) / (2 * a)
        root2 = (-b - discRoot) / (2 * a)
        print("\nThe solutions are:", root1, root2 )
    except ValueError as excObj:
        if str(excObj) == "math domain error":
            print("No Real Roots")
        else:
            print("Invalid coefficient given")
    except:
        print("\nSomething went wrong, sorry!")

main()
```

The multiple `excepts` are similar to `elifs`. If an error occurs, Python will try each `except` in turn looking for one that matches the type of error. The bare `except` at the bottom in this example acts like an `else` and will be used as the default if no previous `except` error type matches. If there is no default at the bottom and none of the `except` types match the error, then the program crashes and Python reports the error.

Notice how I handled the two different kinds of `ValueErrors`. Exceptions are actually a kind of object. If you follow the error type with an `as <variable>` in an `except` clause, Python will assign that variable the actual exception object. In this case, I turned the exception into a string and looked at the message to see what caused the `ValueError`. Notice that this text is exactly what Python prints out if the error is not caught (e.g., `ValueError:  math domain error`). If the exception is not a `ValueError`, this program just prints a general apology. As a challenge, you might see whether you can find an erroneous input that produces the apology.

You can see how the `try...except` statement allows us to write bullet-proof programs. You can use this same technique by observing the error messages that Python prints and designing `except` clauses to catch and handle them. Whether you need to go to this much trouble depends on the type of program you are writing. In your beginning programs, you might not worry too much about bad input; however, professional-quality software should do whatever is feasible to shield users from unexpected results.

7.5 Study in Design: Max of Three

Now that we have decisions that can alter the control flow of a program, our algorithms are liberated from the monotony of step-by-step, strictly sequential processing. This is both a blessing and a curse. The positive side is that we can now develop more sophisticated algorithms, as we did for our quadratic solver. The negative side is that designing these more sophisticated algorithms is much harder. In this section, we'll step through the design of a more difficult decision problem to illustrate some of the challenge and excitement of the design process.

Suppose we need an algorithm to find the largest of three numbers. This algorithm could be part of a larger problem such as determining grades or computing taxes, but we are not interested in the final details, just the crux of the problem. That is, how can a computer determine which of three user inputs is the largest? Here's a simple program outline:

```
def main():
    x1, x2, x3 = eval(input("Please enter three values: "))

    # missing code sets maxval to the value of the largest

    print("The largest value is", maxval)
```

Notice I'm using `eval` as a quick and dirty way to get three numbers; in production code (programs for other users), of course, you should generally avoid `eval`. It's fine here because we are only concerned with developing and testing some algorithm ideas.

Now we just need to fill in the missing section. Before reading the following analysis, you might want to try your hand at solving this problem.

7.5.1 Strategy 1: Compare Each to All

Obviously, this program presents us with a decision problem. We need a sequence of statements that sets the value of `maxval` to the largest of the three inputs, `x1`, `x2`, and `x3`. At first glance, this looks like a three-way decision; we need to execute *one* of the following assignments:

```
maxval = x1
maxval = x2
maxval = x3
```

It would seem we just need to preface each one with the appropriate condition(s), so that it is executed only in the proper situation.

Let's consider the first possibility, that `x1` is the largest. To determine that `x1` is actually the largest, we just need to check that it is at least as large as the other two. Here is a first attempt:

```
if x1 >= x2 >= x3:
    maxval = x1
```

Your first concern here should be whether this statement is syntactically correct. The condition `x1 >= x2 >= x3` does not match the template for conditions shown above. Most computer languages would not accept this as a valid expression. It turns out that Python does allow this *compound condition,* and it behaves exactly like the mathematical relations $x1 \geq x2 \geq x3$. That is, the condition is true when `x1` is at least as large as `x2` and `x2` is at least as large as `x3`. So, fortunately, Python has no problem with this condition.

Whenever you write a decision, you should ask yourself two crucial questions. First, when the condition is true, are you absolutely certain that executing the body of the decision is the right action to take? In this case, the condition clearly states that x1 is at least as large as x2 and x3, so assigning its value to maxval should be correct. Always pay particular attention to borderline values. Notice that our condition includes equal as well as greater. We should convince ourselves that this is correct. Suppose that x1, x2, and x3 are all the same; this condition will return true. That's OK because it doesn't matter which we choose; the first is at least as big as the others, and hence, the max.

The second question to ask is the converse of the first. Are we certain that this condition is true in all cases where x1 is the max? Unfortunately, our condition does not meet this test. Suppose the values are 5, 2, and 4. Clearly, x1 is the largest, but our condition returns false since the relationship $5 \geq 2 \geq 4$ does not hold. We need to fix this.

We want to ensure that x1 is the largest, but we don't care about the relative ordering of x2 and x3. What we really need is two separate tests to determine that x1 >= x2 *and* that x1 >= x3. Python allows us to test multiple conditions like this by combining them with the keyword and. We'll discuss the exact semantics of and in Chapter 8. Intuitively, the following condition seems to be what we are looking for:

```
if x1 >= x2 and x1 >= x3:   # x1 is greater than each of the others
    maxval = x1
```

To complete the program, we just need to implement analogous tests for the other possibilities:

```
if x1 >= x2 and x1 >= x3:
    maxval = x1
elif x2 >= x1 and x2 >= x3:
    maxval = x2
else:
    maxval = x3
```

Summing up this approach, our algorithm is basically checking each possible value against all the others to determine if it is the largest.

With just three values the result is quite simple, but how would this solution look if we were trying to find the max of five values? Then we would need four Boolean expressions, each consisting of four conditions anded together. The complex expressions result from the fact that each decision is designed to stand on its own; information from one test is ignored in the subsequent tests. To see what I mean, look back at our simple max of three code. Suppose the first decision discovers that x1 is greater than x2, but not greater than x3. At this point, we know that x3 must be the max. Unfortunately, our code ignores this; Python will go ahead and evaluate the next expression, discover it to be false, and finally execute the else.

7.5.2 Strategy 2: Decision Tree

One way to avoid the redundant tests of the previous algorithm is to use a *decision tree* approach. Suppose we start with a simple test x1 >= x2. This knocks either x1 or x2 out of contention to be the max. If the condition is true, we just need to see which is larger, x1 or x3. Should the initial condition be false, the result boils down to a choice between x2 and x3. As you can see, the first decision "branches" into two possibilities, each of which is another decision, hence the name *decision tree*. Figure 7.5 shows the situation in a flowchart. This flowchart translates easily into nested if-else statements.

```
if x1 >= x2:
    if x1 >= x3:
        maxval = x1
    else:
        maxval = x3
else:
    if x2 >= x3:
        maxval = x2
    else:
        maxval = x3
```

The strength of this approach is its efficiency. No matter what the ordering of the three values, this algorithm will make exactly two comparisons and assign the correct value to maxval. However, the structure of this approach is more complicated than the first, and it suffers a similar complexity explosion should we try this design with more than three values. As a challenge, you might see if you can design a decision tree to find the max of four values. (You will need if-elses nested three levels deep leading to eight assignment statements.)

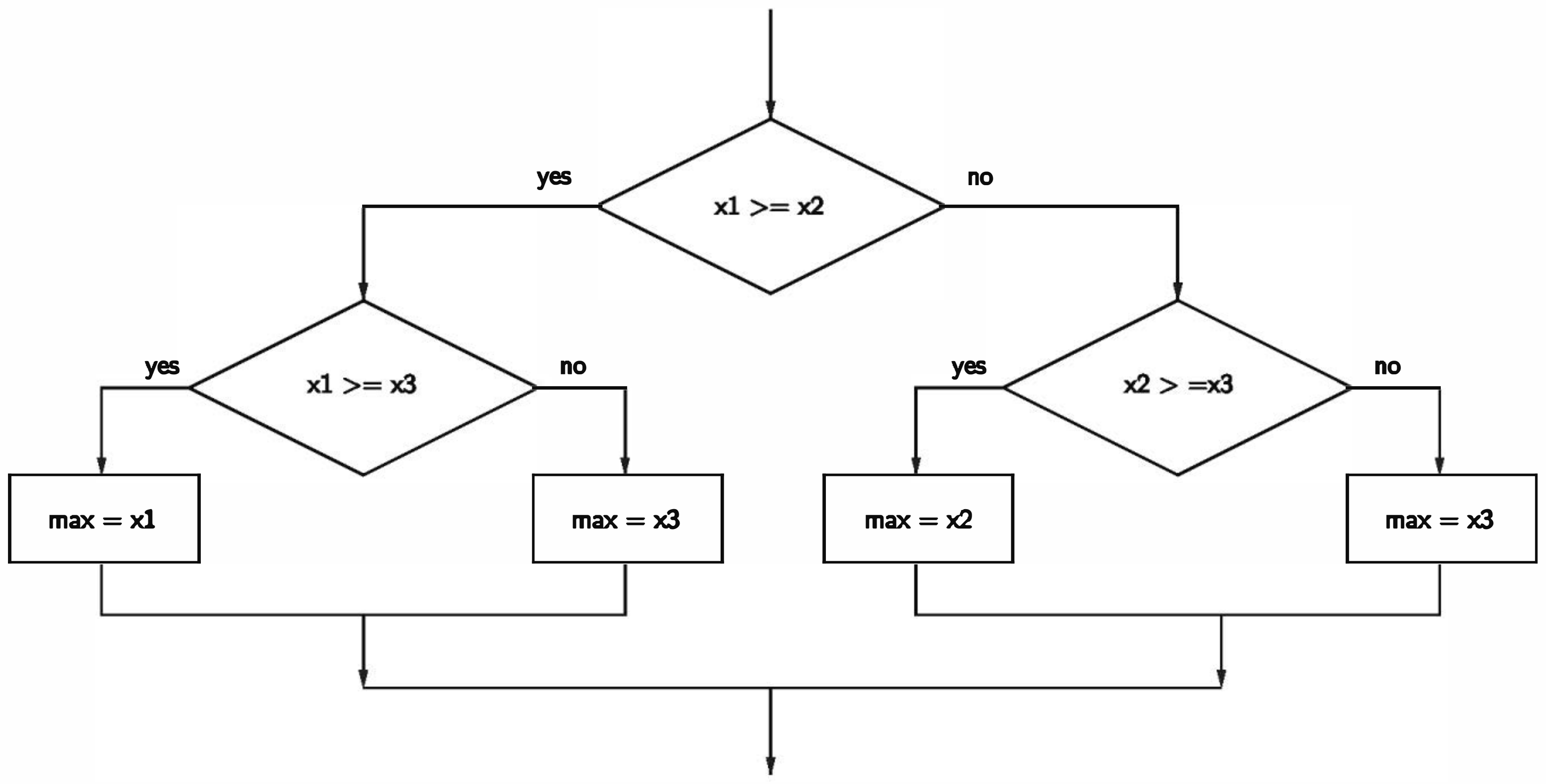


Figure 7.5: Flowchart of the decision tree approach to max of three

7.5.3 Strategy 3: Sequential Processing

So far, we have designed two very different algorithms, but neither one seems particularly elegant. Perhaps there is yet a third way. When designing an algorithm, a good starting place is to ask yourself how you would solve the problem if you were asked to do the job. For finding the max of three numbers, you probably don't have a very good intuition about the steps you go through. You'd just look at the numbers and *know* which is the largest. But what if you were handed a book containing hundreds of numbers in no particular order? How would you find the largest in this collection?

When confronted with the larger problem, most people develop a simple strategy. Scan through the numbers until you find a big one, and put your finger on it. Continue scanning; if you find a number bigger than the one your finger is on, move your finger to the new one. When you get to the end of the list, your finger will remain on the largest value. In a nutshell, this strategy has us look through the list sequentially, keeping track of the largest number seen so far.

A computer doesn't have fingers, but we can use a variable to keep track of the max so far. In fact, the easiest approach is just to use `maxval` to do this job. That way, when we get to the end, `maxval` automatically contains the largest value in the list. A flowchart depicting this strategy for the max of three problem is shown in Figure 7.6.

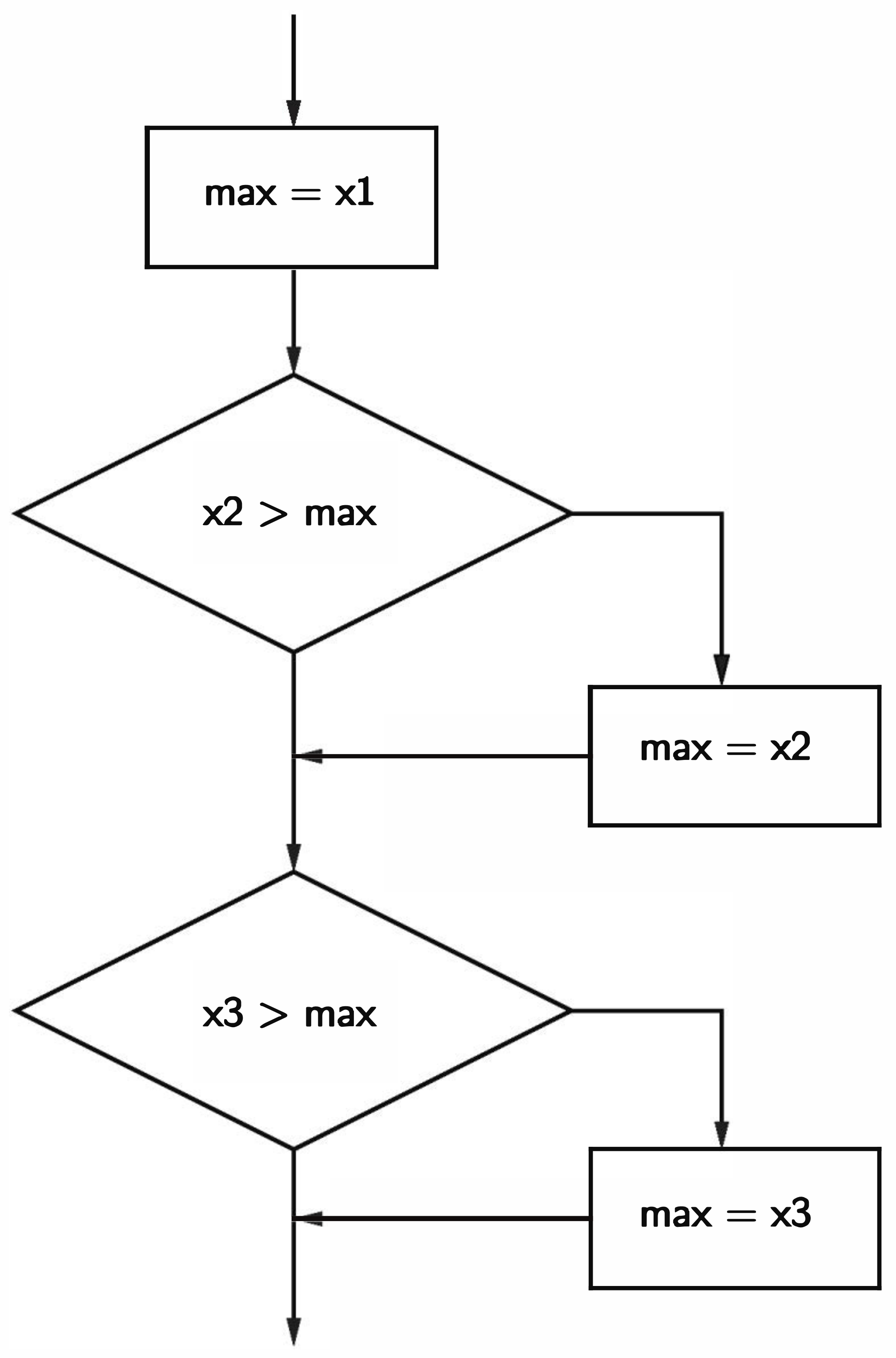


Figure 7.6: Flowchart of a sequential approach to the max of three problem

Here is the corresponding Python code:

```
maxval = x1
if x2 > maxval:
    maxval = x2
if x3 > maxval:
    maxval = x3
```

Clearly, the sequential approach is the best of our three algorithms. The code itself is quite simple, containing only two simple decisions, and the sequencing is easier to understand than the nesting used in the previous algorithm. Furthermore, the idea scales well to larger problems. For example, adding a fourth item requires only one more statement:

```
maxval = x1
if x2 > maxval:
    maxval = x2
if x3 > maxval:
    maxval = x3
if x4 > maxval:
    maxval = x4
```

It should not be surprising that the last solution scales to larger problems; we invented the algorithm by explicitly considering how to solve a more complex problem. In fact, you can see that the code is very repetitive. We can easily write a program that allows the user to find the largest of n numbers by folding our algorithm into a loop. Rather than having separate variables for `x1`, `x2`, `x3`, etc., we can just get the values one at a time and keep reusing a single variable `x`. Each time, we compare the newest `x` against the current value of `maxval` to see if it is larger.

```
# program: maxn.py
#   Finds the maximum of a series of numbers

def main():
    n = int(input("How many numbers are there? "))

    # Set max to be the first value
    maxval = float(input("Enter a number >> "))

    # Now compare the n-1 successive values
    for i in range(n-1):
        x = float(input("Enter a number >> "))
        if x > maxval:
            maxval = x

    print("The largest value is", maxval)

main()
```

This code uses a decision nested inside of a loop to get the job done. On each iteration of the loop, `maxval` contains the largest value seen so far.

7.5.4 Strategy 4: Use Python

Before leaving this problem, I really should mention that none of the algorithm development we have so painstakingly pursued was necessary. Python actually has a built-in function called `max` that returns the largest of its parameters. Here is the simplest version of our program:

```
def main():
    x1, x2, x3 = eval(input("Please enter three values: "))
    print("The largest value is", max(x1, x2, x3))
```

Of course, this version didn't require any algorithm development at all, which rather defeats the point of the exercise! Sometimes Python is just too simple for our own good.

7.5.5 Some Lessons

The max of three problem is not particularly earth shattering, but the attempt to solve this problem has illustrated some important ideas in algorithm and program design.

- There is more than one way to do it. For any non-trivial computing problem, there are many ways to approach the problem. While this may seem obvious, many beginning programmers do not really take this point to heart. What does this mean for you? Don't rush to code up the first idea that pops into your head. Think about your design, ask yourself if there is a better way to approach the problem. Once you have written the code, ask yourself again if there might be a better way. Your first task is to find a correct algorithm. After that, strive for clarity, simplicity, efficiency, scalability, and elegance. Good algorithms and programs are like poems of logic. They are a pleasure to read and maintain.

- Be the computer. Especially for beginning programmers, one of the best ways to formulate an algorithm is to simply ask yourself how you would solve the problem. There are other techniques for designing good algorithms (see Chapter 13); however, the straightforward approach is often simple, clear, and efficient enough.

- Generality is good. We arrived at the best solution to the max of three problem by considering the more general max of n numbers problem. It is not unusual that consideration of a more general problem can lead to a better solution for some special case. Don't be afraid to step back and think about the overarching problem. Similarly, when designing programs, you should always have an eye toward making your program more generally useful. If the max of n program is just as easy to write as max of three, you may as well write the more general program because it is more likely to be useful in other situations. That way you get the maximum utility from your programming effort.

- Don't reinvent the wheel. Our fourth solution was to use Python's `max` function. You may think that was cheating, but this example illustrates an important point. A lot of very smart programmers have designed countless good algorithms and programs. If the problem you are trying to solve seems to be one that lots of others must have encountered, you might begin by finding out if the problem has already been solved for you. As you are learning to program, designing from scratch is great experience. Truly expert programmers, however, know when to borrow.

7.6 Chapter Summary

This chapter has laid out the basic control structures for making decisions. Here are the key points.

- Decision structures are control structures that allow a program to execute different sequences of instructions for different cases.

- Decisions are implemented in Python with `if` statements. Simple decisions are implemented with a plain `if`. Two-way decisions generally use an `if-else`. Multi-way decisions are implemented with `if-elif-else`.

- Decisions are based on the evaluation of conditions, which are simple Boolean expressions. A Boolean expression is either true or false. Python has a dedicated `bool` data type with literals `True` and `False`. Conditions are formed using the relational operators: <, <=, `!=`, ==, >, and >=.

- Some programming languages provide exception handling mechanisms which help to make programs more "bulletproof." Python provides a `try-except` statement for exception handling.

- Algorithms that incorporate decisions can become quite complicated as decision structures are nested. Usually a number of solutions are possible, and careful thought should be given to produce a correct, efficient, and understandable program.

7.7 Exercises

Review Questions

True/False

1. A simple decision can be implemented with an `if` statement.
2. In Python conditions, $\neq$ is written as `/=`.
3. Strings are compared by lexicographic ordering.
4. A two-way decision is implemented using an `if-elif` statement.
5. The `math.sqrt` function cannot compute the square root of a negative number.
6. A single `try` statement can catch multiple kinds of errors.
7. Multi-way decisions must be handled by nesting multiple `if-else` statements.
8. There is usually only one correct solution to a problem involving decision structures.
9. The condition `x <= y <= z` is allowed in Python.
10. Input validation means prompting a user when input is required.

Multiple Choice

1. A statement that controls the execution of other statements is called a
   a) boss structure b) super structure
   c) control structure d) branch
2. The best structure for implementing a multi-way decision in Python is
   a) `if` b) `if-else` c) `if-elif-else` d) `try`
3. An expression that evaluates to either true or false is called
   a) operational b) Boolean c) simple d) compound

4. When a program is being run directly (not imported), the value of `__name__` is
   a) `script` b) `main` c) `__main__` d) `True`

5. The literals for type `bool` are
   a) `T`, `F` b) `True`, `False` c) `true`, `false` d) `1`, `0`

6. Placing a decision inside of another decision is an example of
   a) cloning b) spooning c) nesting d) procrastination

7. In Python, the body of a decision is indicated by
   a) indentation b) parentheses c) curly braces d) a colon

8. A structure in which one decision leads to another set of decisions, which leads to another set of decisions, etc., is called a decision
   a) network b) web c) tree d) trap

9. Taking the square root of a negative value with `math.sqrt` produces a(n)
   a) ValueError b) imaginary number
   c) program crash d) stomachache

10. A multiple choice question is most similar to
    a) simple decision b) two-way decision
    c) multi-way decisions d) an exception handler

Discussion

1. Explain the following patterns in your own words:
   a) simple decision
   b) two-way decision
   c) multi-way decision

2. How is exception handling using `try/except` similar to and different from handling exceptional cases using ordinary decision structures (variations on `if`)?

3. The following is a (silly) decision structure:

```
a, b, c = eval(input('Enter three numbers: '))
```