Chapter 1 Computers and Programs

Objectives

- To understand the respective roles of hardware and software in a computing system.
- 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 or balance your checkbook. 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 and PCs, 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.

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 com-

puter 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?* Put 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 example areas include networking, human-computer interaction, artificial intelligence, compu-

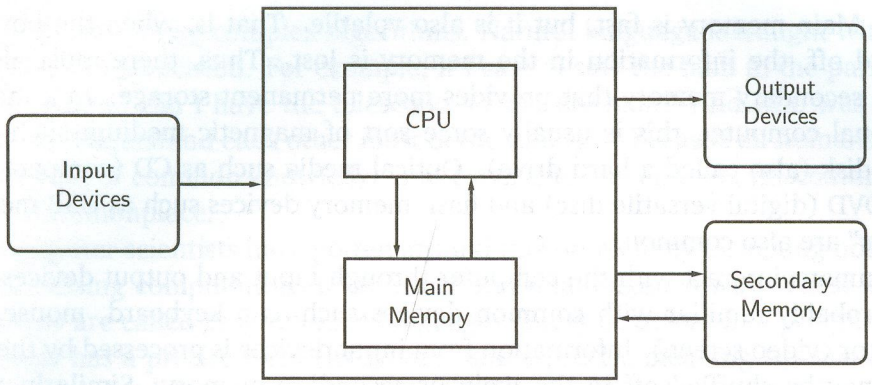

Figure 1.1: Functional view of a computer

tational science (using powerful computers to model scientific data), databases, software engineering, web and multimedia design, 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, etc. 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 only directly access information that is stored in *main memory* (called RAM for *Random Access Mem-*

ory). 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, this is usually some sort of magnetic medium such as a hard disk (also called a hard drive). Optical media such as CD (compact disc) and DVD (digital versatile disc) and flash memory devices such as USB memory "sticks" are also common.

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 millions 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?") Unfortunately, despite the continuing efforts of many top-flight computer scientists (including your author), designing a computer 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. It is the language that we will use throughout this book.[1] You may have heard of some other languages, such as C++, Java, Perl, Scheme, or BASIC. Although these languages differ in many details, they all share the property of having well-defined, unambiguous syntax and semantics. Languages themselves tend to evolve over time.

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 only understand 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:

```
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,

[1] Specifically, the book was written using Python version 3.0. 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.

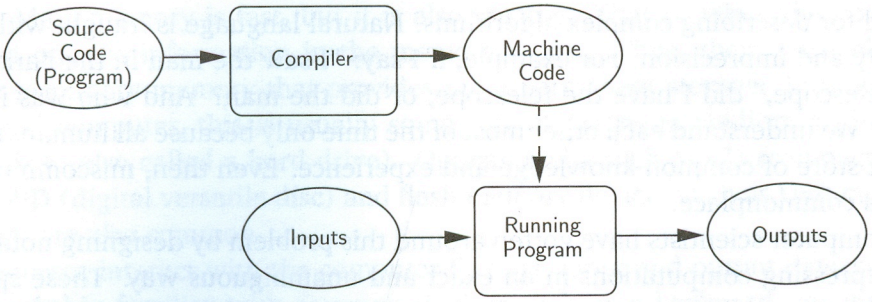

Figure 1.2: Compiling a high-level language

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").

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 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 Core Duo won't run directly on a different CPU. 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

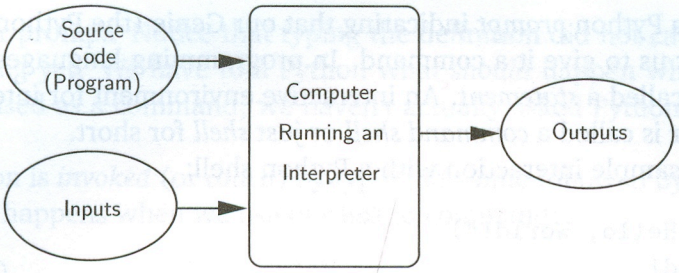

Figure 1.3: Interpreting a high-level language

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 PDA; 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.

You can start the Python interpreter in an interactive mode and type in some commands to see what happens. When you first start the interpreter program, you may see something like the following:

```
Python 3.0 (r30:67503, Jan 19 2009, 09:57:10)
[GCC 4.1.3 20070929 (prerelease) (Ubuntu 4.1.2-16ubuntu2)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

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*. An interactive environment for interacting with an interpreter is called a *command shell* or just *shell* for short.

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 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 on a disk so that it can be used over and over again.

A module file is just a text file, and you can create one using any program for editing text, like a notepad or word processor program (provided you save your program as a "plain text" file). A special type of program known as a *programming environment* simplifies the process. A programming environment is specifically designed to help programmers write programs and includes features such as automatic indenting, color highlighting, and interactive development. The standard Python distribution includes a programming environment called IDLE that you may use for working on the programs in this book.

Let's illustrate the use of a module file by writing and running a complete program. Our program will illustrate a mathematical concept known as chaos. Here is the program as we would type it into IDLE or some other editor and save in a module file:

```
# 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()
```

This file should be saved with the name `chaos.py`. The `.py` extension indicates that this is a Python module. You can see that this particular example contains lines to define a new function 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 in a module file, we can run it any time we want.

This 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 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`. If you are using IDLE (or another programming environment) you can run a program by opening it in the editor and then selecting a command like *import*, *run*, or *execute*.

One method that should always work is to start a Python shell and then `import` the file. Here is how that looks:

```
>>> import chaos
This program illustrates a chaotic function
Enter a number between 0 and 1: .25
0.73125
0.76644140625
0.698135010439
0.82189581879
0.570894019197
0.955398748364
0.166186721954
0.540417912062
```

```
0.9686289303
0.118509010176
>>>
```

Typing the first line `import chaos` tells the Python interpreter to load the `chaos` module from the file `chaos.py` into main memory. Notice that I did not include the `.py` extension on the `import` line; Python assumes the module will have a `.py` extension.

As Python imports the module file, each line executes. 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. When Python encounters the last line of the module, the `main` function is invoked, 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.

When you first import a module file in this way, Python creates a companion file with a `.pyc` extension. In this example, Python creates another file on the disk 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`) file is then interpreted. Having a `.pyc` file available makes importing 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.

A module needs to be imported into a session only once. After the module has been loaded, we can run the program again by asking Python to execute the `main` command. We do this by using a special dot notation. Typing `chaos.main()` tells Python to invoke the `main` function in the `chaos` module. Continuing with our example, here is how it looks when we rerun the program with .26 as the input:

```
>>> chaos.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 `chaos.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 in 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 predict population 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 only understands 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.

- 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