Chapter 1

Computers and Programs

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.

1.1 The Universal Machine

A modern computer might 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 that 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, 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 programs, 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 the masters. 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 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 Edsgar 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?* 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

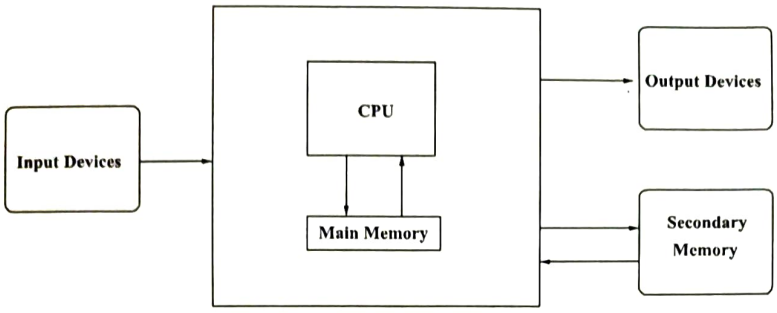

Figure 1.1: Functional View of a Computer.

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.

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 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, this is usually some sort of magnetic medium such as a hard disk (also called a hard drive) or floppy.

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 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. 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.

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 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, 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.

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.

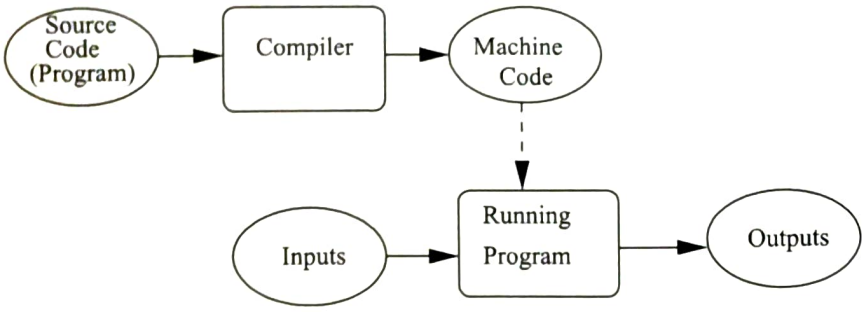

Figure 1.2: Compiling a High-Level Language

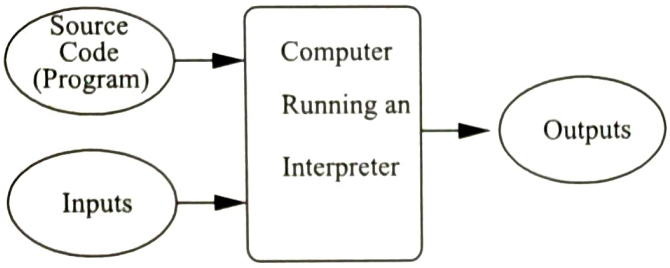

Figure 1.3: Interpreting a High-Level Language.

Each kind of computer has its own machine language. A program for a Pentium CPU won't run on a Macintosh that sports a PowerPC. 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). For example, if I design a new computer, I can also program a Python interpreter for it, and then any program written in Python can be run on my new computer, as is.

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 2.1 (#1, Jun 21 2001, 11:39:00)
[GCC pgcc-2.91.66 19990314 (egcs-1.1.2 release)] on linux2
Type "copyright", "credits" or "license" for more information.
>>>
```

The >>> is a Python *prompt* indicating that the Genie 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 the Python interpreter.

```
>>> 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 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 Python and try the examples for yourself.

Usually we want to move beyond snippets and execute an entire sequence of statements. Python lets us put a sequence of statements together to create a brand-new command called a *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 called `hello`. The following lines are indented to show that they are part of the `hello` function. The blank line (obtained by hitting the <Enter> key twice) lets Python know that the definition is finished, and the interpreter responds with another prompt. Notice that the definition did not cause anything to happen. 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* by typing its name. 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 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* 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 we use `greet` we can send different names to customize the result. We will discuss parameters in detail later on. For the time being, our functions will not use parameters, so the parentheses will be empty, but you still need to include them when defining and using functions.

One problem with entering functions interactively at the Python prompt like this is that the definitions go away when we quit Python. 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 = 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 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 (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 the Python interpreter 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 re-create them as needed.

A module only needs to be imported into a session 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()
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 = 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 (re)run the program by simply invoking `chaos.main()`. We don't have to reload the module from the file 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 = 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 an `input` statement. 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` statement). The computed value is $3.9(0.25)(1 - 0.25)$ or 0.73125. Once the value on the righthand side is computed, it is stored back (or *assigned*) into the variable that appears on the lefthand 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
```

Chapter 2

Writing Simple Programs

As you saw in the previous chapter, it is easy to run programs that have already been written. The hard part is actually coming up with the 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 task. It would be almost impossible without a systematic approach.

2.1 The Software Development Process

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.

Formulate Requirements 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 with 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 if 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.

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. Suzie Programmer has a problem. Suzie is an American computer science student spending a year studying in Europe. 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. Suzie listens to the weather report each morning, but the temperatures are given in degrees Celsius, and she is used to Fahrenheit.

Fortunately, Suzie 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.

Suzie begins with the requirements of her problem. In this case, the problem is pretty clear: the radio announcer gives temperatures in degrees Celsius, but Suzie only comprehends temperatures that are in degrees Fahrenheit. That's the crux of the problem.

Next, Suzie 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. Suzie does some quick figuring to derive the formula $F = (9/5)C + 32$ (Can you see how?). That seems an adequate specification.

Notice that this describes one of many possible programs that could solve this problem. If Suzie had 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 requirements. The purpose of specification is to decide exactly what this particular program will do to solve a problem. Suzie 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.

Suzie 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 convert to a Fahrenheit temperature, and then output the result by displaying it on the computer screen.

Suzie could write her algorithm down in a computer language. However, the precision of writing 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 Suzie'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: Suzie Programmer

def main():
    celsius = input("What is the Celsius temperature? ")
    fahrenheit = 9.0 / 5.0 * 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, Suzie tests it to see how well it works. She uses some 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 Suzie used the values of 0 and 100 to test her program. It looks pretty good, and she is satisfied with her solution. Apparently, no debugging is necessary.

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

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

Table 2.1: Python Reserved Words.

2.3.2 Expressions

Programs manipulate data. The fragments of code that produce or calculate new data values are called *expressions*. So far our program examples have dealt mostly with numbers, so I'll use numeric data to illustrate 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 `9.0`, `5.0`, and `32`. These are all examples of numeric literals, and their meaning is obvious: `32` represents, well, 32.

A simple identifier can also be an expression. We use identifiers as variables to give names to values. When an identifier appears in an expression, this 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 (innermost last):
  File "<pyshell#34>", line 1, in ?
    print spam
NameError: spam
>>>
```

First the variable `x` is assigned the value 5 (using the numeric literal `5`). The next line has Python evaluate the expression `x`. Python spits back `5`, which is the value that was just assigned to `x`. Of course, we get the same result when we put `x` in a print statement. The last example shows what happens when we use a variable that has not been assigned a value. Python cannot find a value, so it reports a *Name Error*. This says that there is no value with that name. 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)
9.0 / 5.0 * celsius + 32
```

Spaces are irrelevant within an expression. The last expression could have been written `9.0/5.0*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 expressions, but you can nest them if necessary to create expressions like this.

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

If you are reading carefully, you may be curious why, in her temperature conversion program, Suzie Programmer chose to write `9.0/5.0` rather than `9/5`. Both of these are legal expressions, but they give different results. This mystery will be discussed in Chapter 3. If you can't stand the wait, try them out for yourself and see if you can figure out what's going on.

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 you can display information on the screen using Python's `print` statement. But what exactly can be printed? Python, like all programming languages, 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 statements.

Here are the possible forms of the `print` statement:

```
print
print <expr>
print <expr>, <expr>, ..., <expr>
print <expr>, <expr>, ..., <expr>,
```

In a nutshell, these templates show that a `print` statement consists of the keyword `print` followed by zero or more expressions, which are separated by commas. The angle bracket notation (<>) is used to indicate "slots" that are filled in by other fragments of Python code. The name inside the brackets indicate what is missing; `expr` stands for an expression. The ellipses ("...") indicate an indefinite series (of expressions, in this case). You don't actually type the dots. The fourth version shows that a `print` statement may be optionally ended with a comma. That is all there is to know about the syntax of `print`.

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

Normally, successive `print` statements will display on separate lines of the screen. A bare `print` (first version above) can be used to get a blank line of output. If a `print` statement ends with a comma (fourth version), a final space is appended to the line, but the output does not advance to the next line. Using this method, multiple `print` statements can be used to generate a single line of output.

Putting it all together, this sequence of `print` statements

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

produces this output

```
7
3 4 7

3 4 7
The answer is 7
```

That last `print` statement may be a bit confusing. According to the syntax templates above, `print` requires a sequence of expressions. That means `"The answer is"` must be an expression. In fact, it *is* an expression, but it doesn't produce a number. Instead, it produces another kind of data called a *string*. A sequence of characters enclosed in quotes is a string literal. Strings will be discussed in detail in a later chapter. For now, consider this a convenient way of labeling output.

2.5 Assignment Statements

2.5.1 Simple Assignment

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. 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 = 9.0 / 5.0 * 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 one 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.

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 special expression called `input`. This template shows the standard form.

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

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

When Python encounters an `input` expression, it evaluates the prompt and displays the result of the prompt on the screen. Python then pauses and waits for the user to type an expression and press the `<Enter>` key. The expression typed by the user is then evaluated to produce the result of the `input`. This sounds complicated, but most uses of `input` are straightforward. In our example programs, `input` statements are used to get numbers from the user.

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

If you are reading programs carefully, you probably noticed the blank space inside the quotes at the end of 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 these two examples specifically prompt the user to enter a number, a number 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 = 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 and stored the value in the variable `ans`. When printed, we see that `ans` got the value 23 as expected.

In a way, the `input` is like a delayed expression. The example interaction produced exactly the same result as if we had simply done `ans = 3 + 4 * 5`. The difference is that the expression was supplied at the time the statement was executed instead of being determined when the statement was written by the programmer. Thus, the user can supply formulas for a program to evaluate.

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:

```
<var>, <var>, ..., <var> = <expr>, <expr>, ..., <expr>
```

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 values 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 = input("Enter two scores separated by a comma: ")
    average = (score1 + score2) / 2.0

    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 using separate input statements.

```
score1 = input("Enter the first score:  ")
score2 = 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 is nice technique to have in your toolkit.

2.6 Definite Loops

You already know that programmers use loops to execute a sequence of statements several 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 from 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 start and end of the body is indicated by its indentation under the loop heading (the `for <var> in <sequence>:` part).

The meaning of the `for` statement is a bit awkward to explain in words, but is very easy to understand, once you get the hang of it. 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. Usually, the `sequence` portion is a list of values. You can build 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
```

You can 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 will execute. 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 which 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. Let's see what the Python interpreter tells us.

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

Do you see what is happening here? The `range` function is a built-in Python command that simply produces a list of numbers. The loop using `range(10)` is exactly equivalent to one using a list of 10 numbers.

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

In general, `range(<expr>)` will produce a list of numbers that starts with 0 and goes up to, but not including, 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 list. In `chaos.py` we did not even care what values the loop index variable used (since the value of `i` was not referred to anywhere in the loop body). We just needed a list of length 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`.

```
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 2.1 depicts the semantics of the `for` loop as a flowchart.

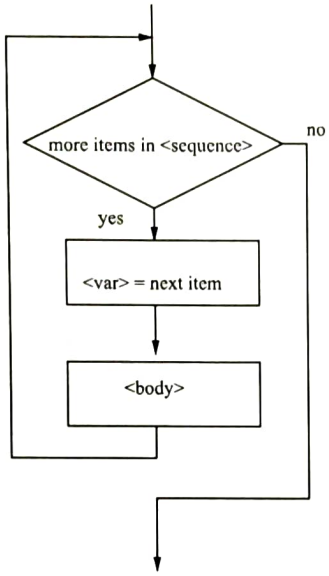

Figure 2.1: Flowchart of a `for` loop.

If you are having trouble understanding the `for` loop, you might find it useful to study the flowchart. The diamond shape box in the flowchart represents a decision in the program. When Python gets the the loop heading, it checks to see if there are any (more) items left if the sequence. If the answer is yes, the value of the loop index variable is set to 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. Let's start with an analysis of the problem (requirements). You know that money that is 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. Recall, 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 annualized percentage rate. Whatever the actual interest rate and compounding frequency, the annualized rate tells us how much the investment accrues in one year. If the annualized interest is 3%, then a

$100 investment will grow to $103 in one year's time. How should the user represent an annualized 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 annualized percentage rate expressed as a decimal fraction.

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 annualized percentage rate (apr)
Repeat 10 times:
    principal = principal * (1 + apr)
Output the value of principal
```

Now that we've thought the problem all the way through to 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 of a 10-year investment"
```

Input the amount of the principal (`input` statement, Section 2.5.2)

```
principal = input("Enter the initial principal:  ")
```

Input the annualized percentage rate (`input` statement, Section 2.5.2)

```
apr = input("Enter the annualized interest rate:  ")
```

Repeat 10 times: (counted loop, Section 2.6)

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

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

```
    principal = principal * (1 + apr)
```

Output the value of the principal (`print` statement, Section 2.4)

```
print "The amount 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
```

```
# by:   John M. Zelle

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

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

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

    print "The amount 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 about it for this example; I leave the testing and debugging as an exercise for you.

2.8 Exercises

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. 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.
4. Modify the `avg2.py` program (Section 2.5.3) to find the average of three exam scores.
5. 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.
6. Modify the `convert.py` program (Section 2.2) with a loop so that it executes 5 times before quitting (i.e., it converts 5 temperatures in a row).
7. 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 0C to 100C.
8. Write a program that converts from Fahrenheit to Celsius.
9. Modify the `futval.py` program (Section 2.7) so that it computes the actual purchasing power of the investment, taking inflation into account. The yearly rate of inflation will be a second input. The adjustment is given by this formula:

```
principal = principal/(1 + inflation)
```