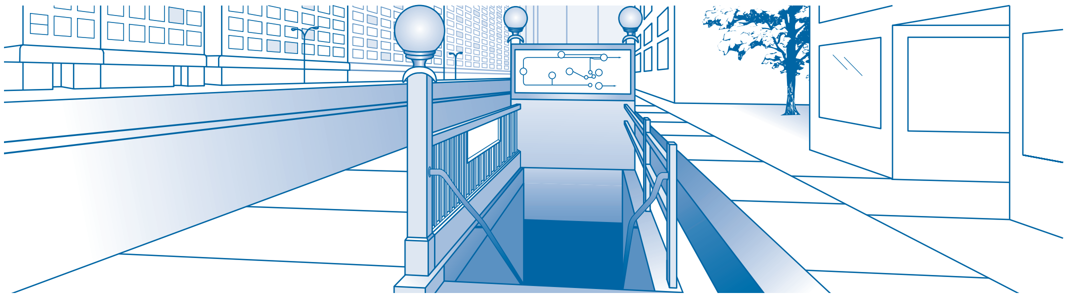
CHAPTER 1

Writing Your First Program

Stations Along the Way

- Installing Python
- Interacting with the Python console
- Building a program that greets the user
- Getting basic text input from the user
- Building a string variable with an appropriate name
- Outputting the value of a string variable to the user
- Creating subsets of a string with slicing
- Using string interpolation to create complex output



Enter the Station

Questions

1. Why is game programming useful?
2. How can game programming prepare you for more “serious” programming study?
3. Why is Python a good language for beginning programmers?
4. What is an interpreted language?
5. What are the advantages and disadvantages of an interpreted language?
6. How do you write the most basic Python program?
7. What is a variable?
8. How do you do basic input and output?
9. How do you name a variable appropriately?
10. How do you modify a string?
11. How do you incorporate variables into output?

You Can Learn a Lot by Writing Games

Welcome to *Game Programming: The L Line*. Over the course of this book, you will learn how to write computer games. You'll learn all the major skills of a game programmer, including how to plan programs, write code, fix mistakes, and manage complicated data.

If you have never written any kind of computer program before, gaming is a great way to start. Learning programming doesn't have to be boring. You'll have a lot of fun because you're going to be building the most interesting kind of program. Game programming provides all the same challenges as other kinds of programming, but games are also visual and interactive. Many programmers were drawn to computers because of games. Learning to program games is even more fun than playing them because you can totally create your own world. Don't worry, though. Even though this book concentrates on games, the skills you learn will translate very well to the more traditional kinds of programming. At the end of this book, you'll be a game programmer, but you'll also be a programmer.

If you've done some programming but you want to learn how to write games, you'll also like this book. A lot of books out there can teach you the mechanics of how to get images on a screen or plot out storyboards, but very few really teach you how to think like a game programmer — and that's my goal. When you imagine a game, I want you to know how to sketch out the relevant ideas, know what kinds of programming constructs you'll need, and have the ability to put your thoughts on the screen in working code. Those are the skills you'll learn in this book. You'll learn some Python, pygame, SDL, and maybe even get a refresher in some basic math and physics.

This book is really about only one thing — writing games. You'll start writing your first game in this chapter, and you'll keep writing games throughout the book. You'll learn how to make text-based games, racing games, arcade games, and even games that involve somewhat sophisticated physics and motion models. There might be some math and theory stuff interspersed, but don't expect a dry textbook. I will tell you about math or physics when it's the solution to a problem, but never for the sake of talking about theory.

Why Use Python?

This book is more about game programming in general than any specific language. When you know how to program games, you can adapt the skills learned here to other languages without too much trouble. In this book, I teach game development by using a language called Python, as well as a specific extension of Python called *pygame*. You have a number of good reasons to use these tools:

- **Python is freely available.** The Python programming language was invented by Guido van Rossum in the early 1990s. He has generously given away the language to anyone who wants it. You don't have to buy Python at all. (Technically, in fact, you *can't* buy it.) A full version of Python is available for download from this book's companion Web site (`www.wiley.com/go/thelline`). When you look at the prices for other programming environments with similar game-development capabilities (Adobe/Macromedia Flash and Microsoft VB.NET come to mind), free looks pretty good. If you add some of the powerful free editing tools and libraries that come with Python, it's even better. Cost is no barrier to programming in Python.
- **Python is platform-independent.** Python works great on a wide variety of computer environments. In this book, I'm assuming you're using Windows, but Python works well on all the primary operating systems in current use, including Mac OS, UNIX, and Linux. The pygame library used for the graphics and games is based on another standard called *SDL* (Simple DirectMedia Layer), which also works on a wide variety of computers. You can rest assured that your games will run on many types of computers.

Transfer

Chapter 4 describes the relationship between pygame and SDL.

- **Python is easy to learn.** One of the original design goals for Python was to make it relatively easy for newcomers to learn. This goal has been largely realized. Python has a reasonably clean syntax (that is, rules of grammar and punctuation) that is not nearly as difficult to learn as many other languages. Writing programming code is still an exacting discipline, but Python does a lot of things automatically for you and makes writing code more straightforward than a lot of other programming languages. For example, many popular programming languages use at least one special character (a semicolon or a brace) at the end of every single line of code. Python uses a different approach, which makes the code easier for beginners to read and write.
- **Python is powerful.** Just because the language is relatively easy, that doesn't mean it's a slouch. Straight Python can do some pretty impressive things. When you attach it to the pygame library (as you will do for most of this book), Python can make some very impressive games. For comparison purposes, most games written in Flash run at 12 to 20 frames per second. Python games typically

run at 30 to 50 frames per second. Although Python itself isn't an exceptionally fast language — compared to compiled languages, such as C — the technique you learn here relies on the very fast SDL library to do most of the heavy lifting. Python also incorporates all the tools needed for many sophisticated techniques, including sprites, data structures, and collision detection.

Transfer

You learn about sprites starting in Chapter 6. You learn about how data is organized beginning in Chapter 2. Collision detection is covered in Chapter 6.

- **Python is extensible.** A lot of great add-on packages are available, and they can extend Python's basic capabilities. In the first section of this book, you will get your hands dirty with the basic version of the Python language. To get to the graphical stuff, you rely on one of Python's best features; the language was designed so people can easily add their own extensions to it, and many people have added wonderful new tools and libraries. The pygame library in particular is a wonderful way to add powerful graphics capabilities to Python. I point out this library and several others as they come up. Of course, I provide all the libraries on the Web site, too, so you can install them for yourself.
- **The Python concepts transfer well to other languages.** Although most commercial games are not written in Python, a good number are — or they use Python as an extension language. Even if you don't end up writing Python games for a living, the basic ideas you learn in this book transfer very well to many other game-development environments.

Installing and Starting Python

Python is pretty easy to install, as programming languages go. In this book, I use version 2.4.2 and 2.5. You can find the Windows Python installer on the companion Web site for this book, or you can go to Python's main page at `www.python.org` and download the latest version for your operating system.

Information Kiosk

Python is a vibrant, changing language. During the time this book was developed, two more minor releases of Python occurred. I've tested the code with 2.4.4, and everything works fine. Any later version of Python should also be fine. Please be sure you're using at least 2.4 or later to ensure all the code in this book works properly.

Installing Python

Install Python to your hard drive by using the standard installation method. The entire language installation weighs in at only 9MB (which is pretty small, considering what it does). After you install it, several new elements appear on the Programs menu, as shown in Figure 1-1.

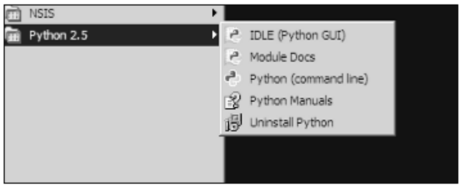
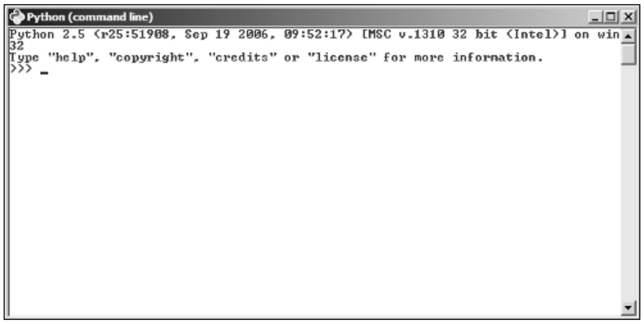


Figure 1-1: The standard version of Python comes with several interesting programs.

The programs that come with Python are pretty handy in their own right. The documentation server (Module Docs) and online help (Python Manuals) both offer a lot of help. Feel free to look at them, but don't get too overwhelmed. There's a lot of information in the documentation that you don't need at first. I show you how to use various documentation features as you need them.

Starting the engine

You can start Python in a couple ways. The simplest is to use the command-line tool, which displays a screen that looks like Figure 1-2. From the Start menu, find the Python group, and choose Python (Command Line).



```
Python 2.5 (r25:51908, Sep 19 2006, 09:52:17) [MSC v.1310 32 bit (Intel)] on win
32
Type "help", "copyright", "credits" or "license" for more information.
>>> _
```

Figure 1-2: It isn't pretty, but it's a good place to start.

If you run the standard Python program from the Windows Start menu, you automatically find yourself at the command-line console for Python. In the Macintosh and Linux environments, the process is slightly different:

1. **Find the command-line console for your operating system.**

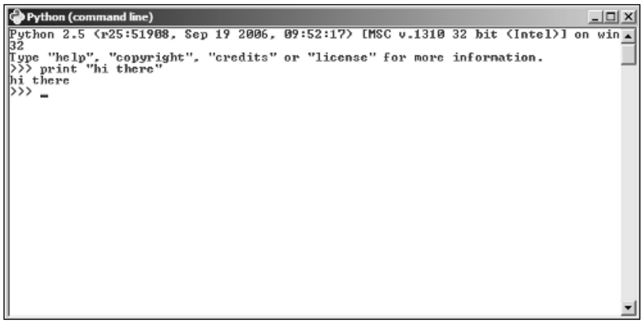
   In most versions of Linux you can right-click the desktop and open a terminal window from the resulting menu.

   On the Mac, open the Terminal program in Applications/Utilities.

2. **Type** python **and press Enter.**

   You'll know you're in the Python environment when you see a screen that looks something like what you see in Figure 1-2. (Each version of Python is slightly different, but they're all similar.)

All versions of Python include this very basic environment, called the *interactive mode.* Although it's nothing to look at, the Python console is surprisingly powerful. Python is an example of an *interpreted* language, which means the programming language is running as you type instructions into it. You can actually type in commands, and Python will respond interactively. For example, type **print "hi, there!"** and see what happens. You should get a screen like the one in Figure 1-3.



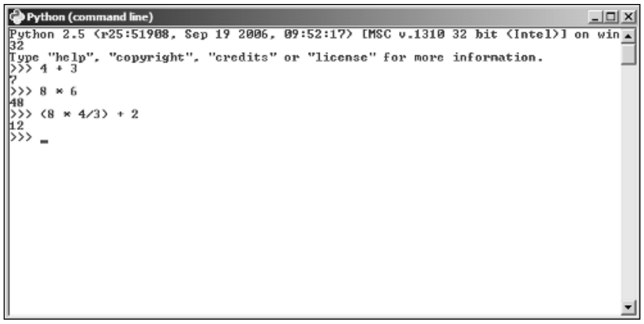
```
Python 2.5 (r25:51908, Sep 19 2006, 09:52:17) [MSC v.1310 32 bit (Intel)] on win
32
Type "help", "copyright", "credits" or "license" for more information.
>>> print "hi there"
hi there
>>> _
```

Figure 1-3: Python is greeting you!

Python is even useful in this very basic mode. Try it out by doing some simple math problems:

1. **Type some sort of arithmetic (4 + 3, for example) after the >>> symbol.**
2. **Press Enter and look at the results.**
3. **To venture into "higher math," use the asterisk (*) for multiplication and the forward slash (/) for division.**
4. **Finish by trying more complex math, including parentheses.**

Figure 1-4 shows the answer for (8 * 4 / 3) + 2. You aren't even halfway through the first chapter, and you've already got something useful!



```
Python 2.5 (r25:51908, Sep 19 2006, 09:52:17) [MSC v.1310 32 bit (Intel)] on win
32
Type "help", "copyright", "credits" or "license" for more information.
>>> 4 + 3
7
>>> 8 * 6
48
>>> (8 * 4/3) + 2
12
>>> _
```

Figure 1-4: Python is a handy calculator.

Information Kiosk

You won't always use the command line to work interactively with Python, because it's a lot easier to type your code into files that are saved on the hard drive. (You learn how to do that later in this chapter.) Still, you'll often find yourself returning to a command-line tool to run little snippets of code in real time so you can see what they do. The interactive capabilities of Python's command console are a nice feature that most modern languages don't have.

Storing information in variables

Python allows you to store information and retrieve it later. To see how this works, do the following:

1. **Type the following code on the command line (the >>> symbol will already be there):**

   ```
   >>> answer = 5 + 3
   ```

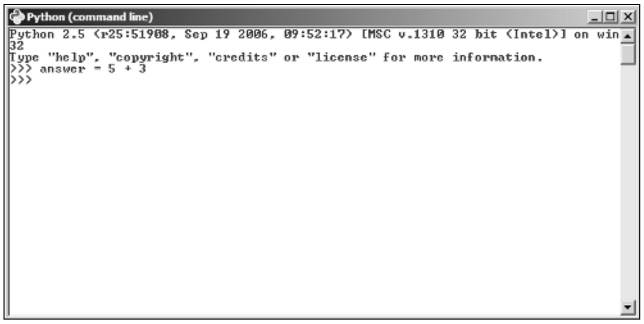
2. **Retrieve the answer by typing in this code:**

   ```
   >>> print answer
   ```

Step Into the Real World

Who's Running What Out There? I'm assuming that most readers use Microsoft Windows as the primary operating system for their computers, but you can write (and run) Python on most other popular operating systems. If you're using a Mac OS X or Linux machine, there's a good chance you already have Python installed. You can find out by going to a command-line console and typing **python**. If you see a message about Python, you're in luck. If you use the Mac OS, check out MacPython at `www.python.org/download/mac`.

When you enter the first line (`answer = 5 + 3`), nothing seems to happen, as shown in Figure 1-5.

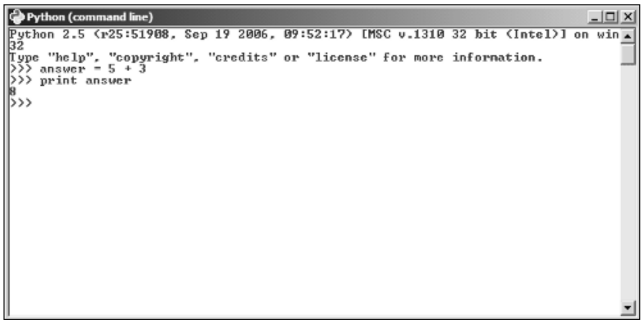


```
Python 2.5 (r25:51908, Sep 19 2006, 09:52:17) [MSC v.1310 32 bit (Intel)] on win
32
Type "help", "copyright", "credits" or "license" for more information.
>>> answer = 5 + 3
>>>
```

Figure 1-5: This command stores information but doesn't display it yet.

Python still calculated the math problem 5 + 3, but rather than having it print out the result, you told it to store the answer in a special element called `answer`. Python automatically created a place in the computer's memory, named it `answer`, and put this value into that place.

After Python prints the answer (`print answer`), you can see from Figure 1-6 that something interesting has happened.



```
Python 2.5 (r25:51908, Sep 19 2006, 09:52:17) [MSC v.1310 32 bit (Intel)] on win
32
Type "help", "copyright", "credits" or "license" for more information.
>>> answer = 5 + 3
>>> print answer
8
>>>
```

Figure 1-6: Now Python prints the results.

You might be surprised that Python prints the number `8` rather than the word `answer`. Quote signs are the secret. If you tell Python to print something inside quotes (such as `"Hello, there!"`), it will print that literal value. If you don't use quotes, Python looks for a place in memory with that name, and prints the resulting value.

These named memory spaces are called *variables*. The ability to store and retrieve information from the computer's memory is one of the key elements in computer programming.

Information Kiosk

When you see a line like `answer = 3 + 5`, train yourself to read it like this: "answer *gets* three plus five." In Python (and many other languages), the equals sign doesn't always mean equality. In other words, it isn't necessarily saying that answer is equal to 3 + 5. Instead, the equals sign is used to indicate *assignment*. In other words, "find the result of the calculation *3 + 5* and store that result in a variable called *answer.*"

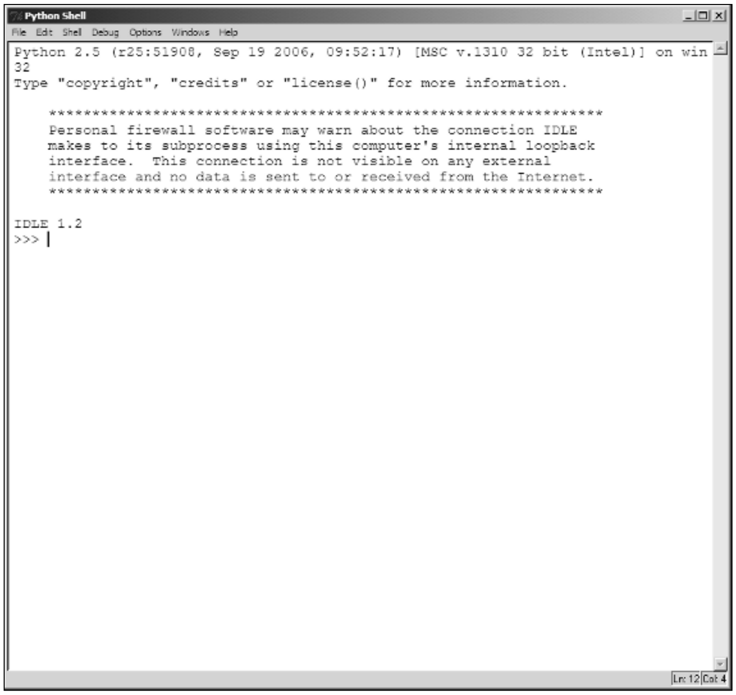
Transfer

In Chapter 3, you'll learn how to test for equality.

Introducing IDLE

The basic Windows and Linux installations of Python come with another nice tool for writing Python programs called IDLE. IDLE is a somewhat farfetched acronym for Interactive DeveLopment Environment. (I guess if you're smart enough to make your own programming language, you can capitalize any letter you want.)

The IDLE environment is featured in Figure 1-7.



```
Python 2.5 (r25:51908, Sep 19 2006, 09:52:17) [MSC v.1310 32 bit (Intel)] on win
32
Type "copyright", "credits" or "license()" for more information.

    ****************************************************************
    Personal firewall software may warn about the connection IDLE
    makes to its subprocess using this computer's internal loopback
    interface.  This connection is not visible on any external
    interface and no data is sent to or received from the Internet.
    ****************************************************************

IDLE 1.2
>>> |
```

Figure 1-7: IDLE is a little easier on the eyes than the command-line console.

Step Into the Real World

IDLE is an example of an Integrated Debugging Environment (IDE,) which is just a fancy term for a text editor optimized to work with a programming language. You can use any text editor you want to make Python programs, but IDLE (or any other IDE meant to work with Python) will make the job easier. IDLE is usually packaged with the Windows and Linux versions of Python. If you are using the MacPython environment, there is an IDE called (cleverly enough) pythonIDE, which does pretty much the same thing as IDLE. If you don't have IDLE or another IDE, you can still do everything. Simply write your programs in a text editor and use `python filename.py` from the command line to run the program. Use python without any filename to run Python in interactive mode.

IDLE has some nice features that make it an attractive tool for building Python programs. It acts like an interactive text editor that understands Python syntax. To illustrate, try this out:

1. **Start IDLE as you start any program in your operating system.**

   You can run it from the Start menu in Windows, by double-clicking the IDLE icon, or by typing `idle` into your operating system's command-line console.

2. **Notice the standard prompt.**

   When Python is waiting for you to do something, it presents a standard >>> symbol. You saw this same symbol when you ran the more primitive console version of Python.

3. **Look at the menus.**

   Unlike the standard Python console, IDLE features menus with various commands on them. Don't worry if you don't know what these mean yet. Notice there is a Help menu. That will probably come in handy soon.

4. **Type in a command.**

   Try typing the following line into the IDLE console:

   ```
   print ("hi there")
   ```

   It's important that you type exactly what you see here, including the parentheses and the quotes, or Python will not respond correctly.

5. **Press Enter to tell Python to execute the command.**

   IDLE will respond much like the command line did, printing out "hi there."

6. Notice that the color changes.

You'll see something interesting when you type commands into IDLE. The word `print` automatically appears in red, and the text `"hi there"` will be colored green. This happens because IDLE recognizes `print` as a built-in command, and `"hi there"` as the thing it should print. This feature is called *syntax highlighting*. It can make things much easier on you as you write your code.

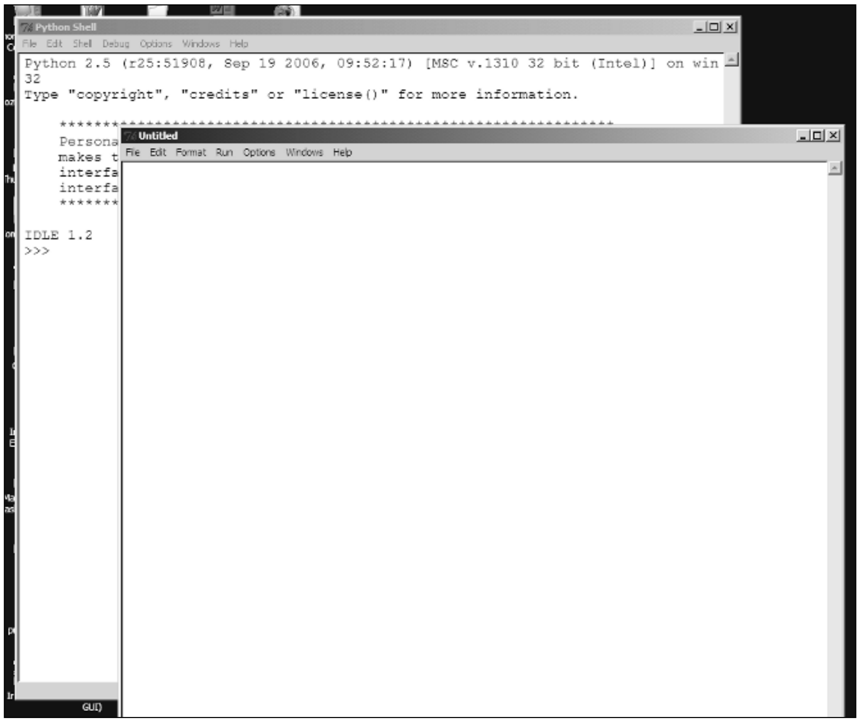
IDLE features a number of other interesting tools that I'll point out as you continue your travels.

Storing your code in a file

It's convenient to write commands in interactive mode, but you'll soon want to write programs that are more than one command long. Python allows you to write a whole slew of commands in a text file and run them all at one time. IDLE has a built-in text editor that is perfect for storing all your commands. To use it, follow these steps:

1. Open a new window.

From the File menu, choose the New Window command. Figure 1-8 shows the original IDLE window with a new window on top.


Python Shell
File Edit Shell Debug Options Windows Help
Untitled
File Edit Format Run Options Windows Help

```
Python 2.5 (r25:51908, Sep 19 2006, 09:52:17) [MSC v.1310 32 bit (Intel)] on win
32
Type "copyright", "credits" or "license()" for more information.

    ****************************************************************
    Persona
    makes t
    interfa
    interfa
    *******

IDLE 1.2
>>>
```

Figure 1-8: IDLE after opening a new file window.

2. See how the new editor window is different.

The new window looks a lot like the original, but there are some subtle differences. First, note that the new window is blank. It does not have the >>> prompt you see when you run Python in interactive mode. Also, the menu items are slightly different. This second window is a special text editor used for writing Python commands.

3. Enter a command into the window.

Type the following command into the new window:

```
print "Hello, World! "
```

When you press the Enter key, you might be surprised. In the standard IDLE window, everything happens interactively. As soon as you type a command, Python immediately responds, but when you use IDLE as a text editor, nothing seems to happen. The editor is letting you enter a command (or a series of commands), but the editor does not execute the commands until you tell it to. Writing commands in this way is a lot like writing a recipe. You're writing down the steps, but you aren't actually cooking (executing the commands) yet.

4. Save your file.

Use the Save command from the File menu to save your masterpiece as `Hello.py`.

Information Kiosk

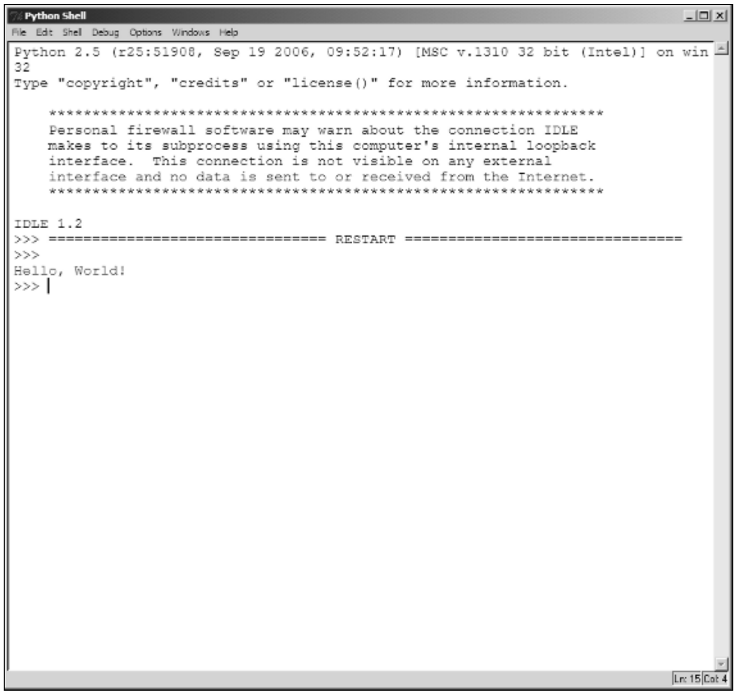
It's important that you use the `.py` extension, because that's how your computer knows that the text file you're saving is actually a Python program and not some other text file (like a shopping list or something).

5. Run your program.

From the Run menu, select the Run Module command (or just press F5) to run your program. You should get a screen that looks like Figure 1-9:

Step Into the Real World

IDLE is the only editor you need to write all the programs in this book. However, you don't *have* to use it. You can actually write Python code in any text editor you wish, as long as it saves code in plain text format. (Word processors do not do this well because they write documents in specialized formats that the Python interpreter cannot need. Word processors also emphasize special formatting and layout features that are meaningless for a programming editor.) For now, stick with IDLE if you can, because you already have it, and it does plenty. As the programs get more complicated, you might want to use a more complex editor. I'll describe one such editor (called SPE) later on in Chapter 5.



```
Python 2.5 (r25:51908, Sep 19 2006, 09:52:17) [MSC v.1310 32 bit (Intel)] on win
32
Type "copyright", "credits" or "license()" for more information.

    ****************************************************************
    Personal firewall software may warn about the connection IDLE
    makes to its subprocess using this computer's internal loopback
    interface.  This connection is not visible on any external
    interface and no data is sent to or received from the Internet.
    ****************************************************************

IDLE 1.2
>>> ================================ RESTART ================================
>>>
Hello, World!
>>> 
```

Figure 1-9: Now the program is running!

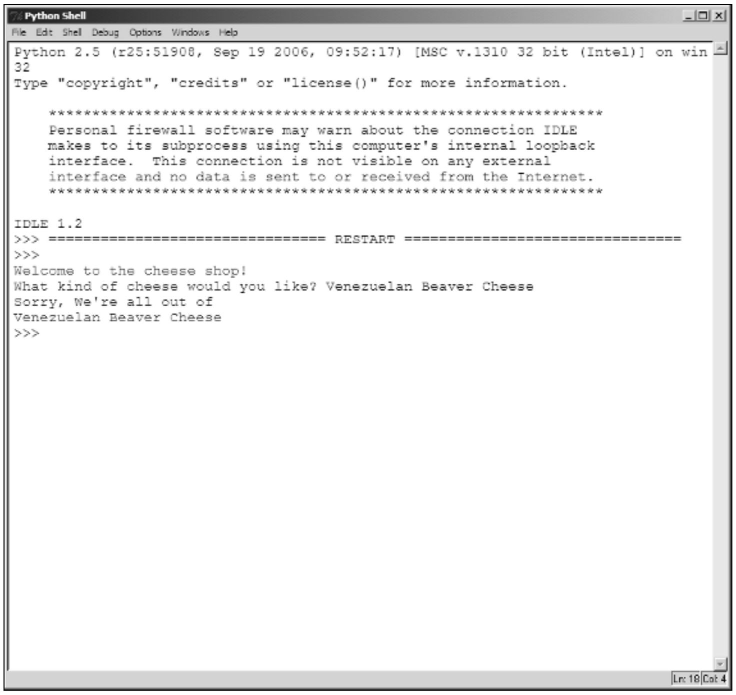
When you run a program from the editor screen, Python takes all the commands from the file and executes them all at once. For these early programs, you'll see the results in the main IDLE screen. Later on, your programs will build their own output window.

Information Kiosk

If you use another text editor (one that doesn't have a "run Python" command), be sure to save the file with a `.py` extension. If you double-click on the file, the operating system can usually run it by associating it with the Python interpreter. However, I don't recommend you do it this way at first, because the program will disappear as soon as it finishes, and you won't be able to see it. If you cannot use IDLE (because you're working in Linux, for example), go to the command line and run your program directly from the command console by entering the following: `python Hello.py`.

Writing Your First Interactive Game

It's pretty interesting to print something to the screen, but it'd be even better if you could make a program that acts a little more like a game. Check the Cheese Shop game shown in Figure 1-10 for a silly example.



```
Python 2.5 (r25:51908, Sep 19 2006, 09:52:17) [MSC v.1310 32 bit (Intel)] on win
32
Type "copyright", "credits" or "license()" for more information.

    ****************************************************************
    Personal firewall software may warn about the connection IDLE
    makes to its subprocess using this computer's internal loopback
    interface.  This connection is not visible on any external
    interface and no data is sent to or received from the Internet.
    ****************************************************************

IDLE 1.2
>>> ================================ RESTART ================================
>>>
Welcome to the cheese shop!
What kind of cheese would you like? Venezuelan Beaver Cheese
Sorry, We're all out of
Venezuelan Beaver Cheese
>>>
```

Figure 1-10: There's not much cheese at this cheese shop.

Take a look at all of the following code. Notice that it is almost English-like. Even though you may not understand everything immediately, Python code is pretty straightforward, and you'll probably be able to guess what's going on. Still, I'll explain every single part of this program after you look it over.

Here is the code for the `cheeseShop.py` program:

```
"""Cheese Shop
    cheeseShop.py
    demonstrate comments
    raw input,
    and string variables
    from Game Programming - L-line
    Andy Harris
    4/10/06
    """

#tell the user something
print "Welcome to the cheese shop!"

#get information from the user
cheeseType = raw_input("What kind of cheese would you like? ")

#we don't have that kind...
print "Sorry, We're all out of"
print cheeseType
```

Step Into the Real World

The name of the Python language is a tribute to the British comedy troupe Monty Python. The `cheeseShop.py` program is inspired by one of their more famous skits. If you're going to program in this language, you may have to get some old episodes from the library. For some strange reason, computer programmers tend to love the odd humor of this group. I have to admit, I'm a big fan of them myself. You might just see another reference to them as we go on . . . The `cheeseShop.py` program welcomes the user and asks what type of cheese he would like. Regardless of the cheese requested, the program politely informs the user that no such cheese is available.

This program shows some important new ideas: documentation strings, comments, input, and output. Don't worry. None if it is all that hard to understand.

Using docstrings to document your code

Take a close look at the first few lines of the `cheeseShop.py` program. They begin and end with a triple double-quote symbol (`"""`). This special combination indicates multiple lines of text. If you have several lines of text embedded inside the triple double-quotes at the beginning of the program, this text will be used as online help for the program. Each program should begin with at least the same kind of information I indicated for the `cheeseShop.py` program:

- The name of the program
- The filename where the program is stored
- The author's name
- The date the program was written or modified
- What the program does

This special documentation is called a *docstring.* Although Python ignores the information in this text block, it is very useful for human programmers. There's also an important system called `pydoc` that uses these documentation values to automatically build online help for your programs. Once programs get long and complicated, you will really appreciate your habit of good documentation. You'll be amazed at how little you understand a program you wrote after only a couple of weeks have passed. However, when you develop good documentation habits, your programs will be readily understandable because you'll leave reminders to yourself or other programmers about what the code is and does.

Building comments

The next line of code is also ignored!

Step Into the Real World

Documentation is a Part of Python One of the fine traditions of Python is how well the code is documented. Most programming languages have completely separate Help systems. When you install the language, you must also install Help files so you will know how everything works. The Help system in these languages is not always completely up to date, because the Help system is often updated months after changes are made to the underlying code. In Python, the Help system is built directly into the code, so if you have a chunk of Python code, you already have the relevant documentation. When you write your own programs, you have an obligation to continue that tradition by placing appropriate documentation inside your programs, as I have done in the `cheeseShop.py` program.

```
#tell the user something
```

This command begins with a pound sign (#). Inside your code, you can add more comments by beginning a line with this symbol. In this first program, I added a comment to explain every single line of code. As you get more sophisticated, you won't need that many comments. It's still a great idea to sprinkle comments into your code when you write something clever. (Clever code means you won't understand it in a couple weeks.)

Printing output to the screen

After taking care of documentation strings and comments, we get to a line that actually does something:

```
print "Welcome to the cheese shop!"
```

You've actually used the `print` command a couple times already. This command does what it says: It prints something to the screen. The text inside the quotation marks is the stuff that Python will print out. Commands in Python (or any other programming language) are sometimes called *statements.* Some statements (such as `print`) accept some value to work with. This value is often called the *argument,* so the argument of the `print` statement above is `"Welcome to the cheese shop!"`

Watch Your Step

You must use the standard quote symbols in Python. Word processors love to put in *"smart quotes,"* which have different symbols for the beginning and end of a quote. (Look at the preceding sentence for an example of smart quotes.) Python will be confused by these characters. To avoid this headache, simply don't use a word processor to write your programming code. IDLE or another plain-text editor will produce the kind of text you need.

When Python interprets the `print` command, it will print out whatever text is inside the quote symbols. If you don't use quote signs, Python expects the name of a variable, and it will print out the value of that variable. More on that in a moment.

Getting input from the user

The next line of our `cheeseShop.py` program is really interesting:

```
cheeseType = raw_input("What kind of cheese would you like? ")
```

The `raw_input()` function allows you to get input from the user. The function requires a prompt — in this case, a question that Python asks the user: `"What kind of cheese would you like?"` The program then stops and lets the user type a value into the command console. When the user presses Enter, whatever he or she typed is transferred to the `cheeseType` variable.

Transfer

In this chapter you're learning how to use functions that come with Python, but in Chapter 3 you learn how to create and use your own functions.

Information Kiosk

If you've already programmed in other languages, you might be surprised that you don't have to create or declare a variable explicitly. When you refer to the `cheeseType` variable, Python automatically creates the variable and figures out what type it should be. Python supports a feature called Implicit Variable Declaration — which means that simply mentioning a variable creates it. The good news is it is very easy to create a variable in Python. There is some bad news: If you misspell a variable name, Python just assumes you wanted to create a *new* variable. Also, Python guesses about what type of data you intend the variable to hold — and it sometimes guesses wrong.

Creating a variable

Variables — those handy places to put things — are one of the most important parts of computer programming in any language. You will create a lot of variables as you become a programmer. It's a good idea to learn how variables should be named, because you get to name a lot of things in programming. Variable names should follow these basic rules:

- **Be descriptive.** A variable's name should indicate what the variable means. The `cheeseType` variable used in this example is well-named because it is clear that this variable is supposed to hold a type of cheese. `R` or `p` are not good variable names, because it's hard to say exactly what they mean. Generally your variable names should be long enough to clearly indicate their meaning.

- **Use single words.** Variable names cannot contain spaces. If you want to make a variable name out of two different words (such as *cheese type*), you have two main choices: You can use the underscore character (_), as in `cheese_type`, or you can use what's called "mixed case," as in `cheeseType`. Note that only the `T` in `Type` is capitalized. Throughout this book, I use mixed case to create variable names.
- **Be case-sensitive.** `CHEESETYPE`, `cheeseType`, and `ChEeSeTyPe` are three totally different variables as far as Python is concerned. Python is a case-sensitive language — meaning you can't get sloppy about when you use uppercase and lowercase characters in your variables or commands. It's best to stick with mixed-case to avoid problems.
- **Be manageable.** Variable names should be long enough to be readable, but not so long that you can no longer type them. For example, `variableThatHoldsTheNameOfATypeOfCheese` is too long a variable name. You'll have trouble spelling it correctly, and it will annoy you.

Information Kiosk

The `raw_input()` function has another powerful use. If you've tried running your programs by double-clicking them from Windows, you'll find they close immediately after completion. Put the following code at the end of your program to make it wait long enough to read:

```
raw_input("press Enter to continue")
```

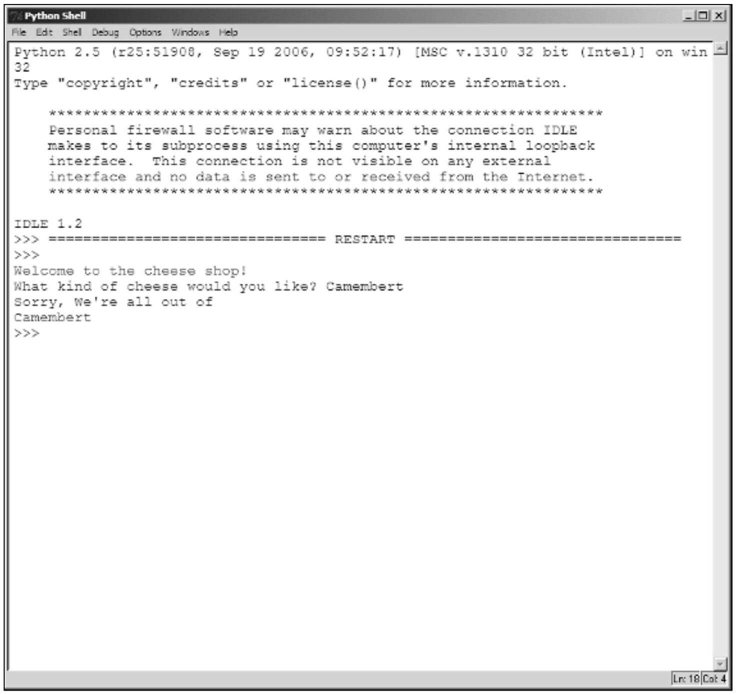
This command prints out the prompt, and then it waits. In this case, it doesn't bother to save the result to a variable, because you don't really care what the user typed. You're just waiting for a press on the Enter key to know the user is finished. You won't need this trick if you test your code within IDLE, but it can be handy.

Printing the results

The last two lines of code print both a text literal value and the value of the variable `cheeseType`:

```
#we don't have that kind...
print "Sorry, We're all out of"
print cheeseType
```

Figure 1-11 illustrates how this code prints the two different kinds of values.



```
Python 2.5 (r25:51908, Sep 19 2006, 09:52:17) [MSC v.1310 32 bit (Intel)] on win
32
Type "copyright", "credits" or "license()" for more information.

    ****************************************************************
    Personal firewall software may warn about the connection IDLE
    makes to its subprocess using this computer's internal loopback
    interface.  This connection is not visible on any external
    interface and no data is sent to or received from the Internet.
    ****************************************************************

IDLE 1.2
>>> ================================ RESTART ================================
>>>
Welcome to the cheese shop!
What kind of cheese would you like? Camembert
Sorry, We're all out of
Camembert
>>>
```

Figure 1-11: The program prints a literal value and a variable.

When Python encounters a phrase inside quotes, it prints that phrase exactly (which is why such a value is called a *literal*). The first `print` statement is that kind of command, so Python simply prints its value.

The second `print` statement doesn't have quotes, so Python looks for a variable named `cheeseType`. If it can find such a variable, it prints the corresponding value.

Watch Your Step

When asked to print a word with no quotes, Python looks for a variable with that name and prints the value of that variable. If no such variable exists, Python will quietly create an empty variable. Be very careful with your spelling, because if you misspell a variable name, Python will complain and the program won't run.

Introducing String Variables

Many of the variables you encounter in this book contain text. These variables are called *string* variables, because the way each character's data was stored in contiguous cells reminded the early programmers of beads on a string. It's oddly poetic, and the term has stuck. Now that you're an official programmer, you should start referring to text as "string data." People will be impressed with your intellectual prowess.

Building a basic string

Begin experimenting with string values by typing this command into the console.

1. **Type a string assignment into the console:**

   ```
   playerName = "Princess Oogieboogie"
   ```

   Python automatically creates a variable called `playerName`. Since `"Princess Oogieboogie"` is a text value, `playerName` is magically made into a string variable, and it can do all the things that strings can do in Python.

2. **Print out the value of the string:**

   ```
   >>> print playerName
   Princess OoogieBoogie
   ```

Information Kiosk

It's not really magic, of course. Python is an example of a *dynamically typed language*, which means that the programmer doesn't have to explicitly say what kind of data goes into a variable. The language guesses for you. If you know anything about computers, you know sometimes it guesses wrong. Don't worry. I'll show you how to recognize and prevent such problems as you move along.

Using string methods

Python supports a special form of programming called *object-oriented programming* (OOP). This kind of programming describes various things as *objects*. Strings are the first objects you encounter, but there are many more. You will actually build objects as the primary actors in your arcade games. (You'll get there soon, I promise.) Objects are nice, because they already know how to do things. To demonstrate, try this little exercise:

1. **Create a string variable. Type the following in interactive mode (that is, using the command line or IDLE's own interactive mode, but not the text editor — you want this code to execute immediately):**

   ```
   >>> king = "Arthur"
   ```

2. **Print the uppercase version of the variable. Type this line into the console:**

   ```
   >>> print king.upper()
   ARTHUR
   ```

Step Into the Real World

Methods Are Things that Objects Can Do. If you had a `monkey` object, you might expect it to have methods like `dance()`, `screech()`, and `throwBanana()`. Method names usually end in parentheses, as I have done here. The great thing about using objects is you don't have to understand how the method works. If you have a monkey, you can make it throw bananas by invoking the `monkey.throwBanana()` method. Likewise, you can get the uppercase version of a string called `king` by invoking `king.upper()`.

Python prints `ARTHUR` (all in uppercase). When you assigned a text value to the variable `king`, you created a string variable. String variables come with all kinds of cool capabilities, called *methods*. The `upper()` method makes a new version of the string all in uppercase. Note that `king.upper()` doesn't change the underlying value of the `king` variable. It simply returns a version of `king` formatted all in uppercase.

Transfer

For more information on IDLE and how to debug in it, turn to Chapter 2.

Examining other string capabilities

String variables have lots of other cool methods. To learn some of what a string can do, try out these steps:

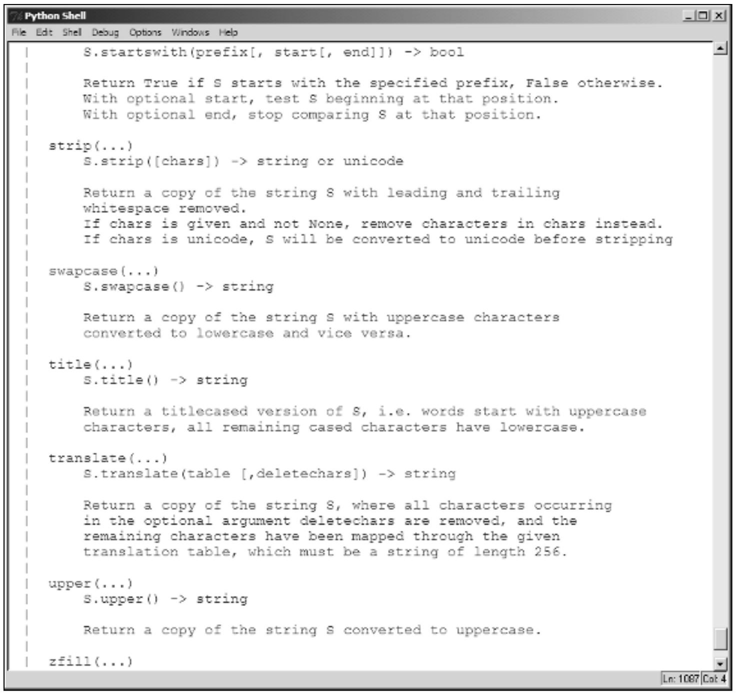
1. **Ask Python for help. Go back to the Python console and type**

   ```
   help("str")
   ```

2. **Read all about strings. Python responds with a huge amount of information, as shown in Figure 1-12.**

Information Kiosk

You don't have to understand all these methods now. It's good to know that they are there, and you can find them if you need them. The much more important lesson here is how the `help()` function works. You use the `help()` function with any Python object or command to find out lots of useful information about that object.



```
 |      S.startswith(prefix[, start[, end]]) -> bool
 |
 |      Return True if S starts with the specified prefix, False otherwise.
 |      With optional start, test S beginning at that position.
 |      With optional end, stop comparing S at that position.
 |
 |  strip(...)
 |      S.strip([chars]) -> string or unicode
 |
 |      Return a copy of the string S with leading and trailing
 |      whitespace removed.
 |      If chars is given and not None, remove characters in chars instead.
 |      If chars is unicode, S will be converted to unicode before stripping
 |
 |  swapcase(...)
 |      S.swapcase() -> string
 |
 |      Return a copy of the string S with uppercase characters
 |      converted to lowercase and vice versa.
 |
 |  title(...)
 |      S.title() -> string
 |
 |      Return a titlecased version of S, i.e. words start with uppercase
 |      characters, all remaining cased characters have lowercase.
 |
 |  translate(...)
 |      S.translate(table [,deletechars]) -> string
 |
 |      Return a copy of the string S, where all characters occurring
 |      in the optional argument deletechars are removed, and the
 |      remaining characters have been mapped through the given
 |      translation table, which must be a string of length 256.
 |
 |  upper(...)
 |      S.upper() -> string
 |
 |      Return a copy of the string S converted to uppercase.
 |
 |  zfill(...)
```

Figure 1-12: Here are some of the things string objects can do.

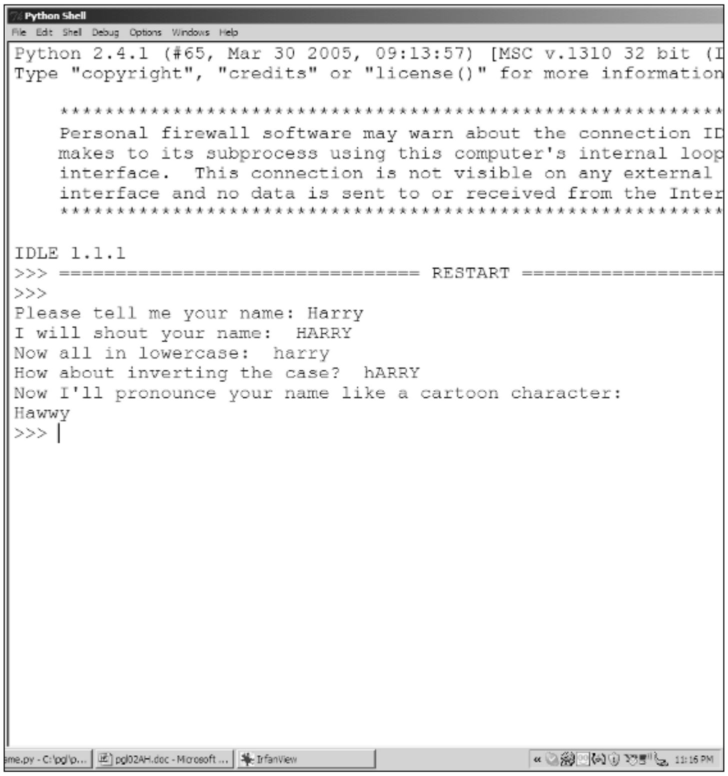
Building a name game with string manipulation

For an example of the power of Python strings, look at the following `nameGame` code. It uses a number of the string methods you saw when you asked Python what strings can do.

```
""" nameGame.py
    illustrate basic string functions
    Andy Harris
    3/15/06"""

userName = raw_input("Please tell me your name: ")
print "I will shout your name: ", userName.upper()
print "Now all in lowercase: ", userName.lower()
print "How about inverting the case? ", userName.swapcase()
numChars = len(userName)
print "Your name has", numChars, "characters"
print "Now I'll pronounce your name like a cartoon character:"
userName = userName.upper()
userName = userName.replace("R", "W")
userName = userName.title()
print userName
```

The nameGame.py code produces the output shown in Figure 1-13.



```
Python 2.4.1 (#65, Mar 30 2005, 09:13:57) [MSC v.1310 32 bit (I
Type "copyright", "credits" or "license()" for more information

    ****************************************************************
    Personal firewall software may warn about the connection ID
    makes to its subprocess using this computer's internal loop
    interface.  This connection is not visible on any external
    interface and no data is sent to or received from the Inter
    ****************************************************************

IDLE 1.1.1
>>> ================================ RESTART ==================
>>>
Please tell me your name: Harry
I will shout your name:  HARRY
Now all in lowercase:  harry
How about inverting the case?  hARRY
Now I'll pronounce your name like a cartoon character:
Hawwy
>>> |
```

Figure 1-13: Python allows you to change strings in interesting ways.

Choosing your string methods

To build the name game, I simply looked at the various string methods and played around with them to see what I could do. Python has some fun string-manipulation commands. Table 1-1 lists the ones I used in this program.

Table 1-1 Selected String Methods

String Method	Description
`stringName.upper()`	Converts *stringName* into all uppercase
`stringName.lower()`	Converts *stringName* into all lowercase
`stringName.swapcase()`	Converts uppercase to lowercase, lowercase to uppercase

String Method	Description
`stringName.replace(old, new)`	Looks in the string for the value *old* and replaces it with the value *new*
`stringName.title()`	Capitalizes each word in the string
`len(string)`	Returns the length of the string

Information Kiosk

There are many more string methods available. I demonstrate more as they come up, but don't feel like you have to memorize them. You can always look up the details in the online help. The important thing is to know the kinds of methods that are available so you'll know to look them up when the time comes.

Determining the length of the name

Sometimes you might want to know the number of characters in a phrase. The `len()` function can be used for this.

```
numChars = len(userName)
print "Your name has", numChars, "characters"
```

Information Kiosk

The `len()` function is technically a method, even though it works somewhat differently from the other methods. The actual method is `__len__()`. Python allows methods defined like this to be treated as functions, so you can use `numChars = len(userName)` to retrieve the length of `userName` and store it into the variable `numChars`.

Making the cartoon version

The last few lines of the name game are a little bit interesting. See if you can figure out what I did, and why:

```
userName = userName.upper()
userName = userName.replace("R", "W")
userName = userName.title()
```

The main thing I wanted to do here was simulate cartoon speech by replacing `R` with `W`. I didn't want to worry about replacing both upper- and lowercase `R` values, so I did something sneaky. (Perhaps I'm a wascally wabbit.) Here's how to convert the name to a cartoon format:

1. **Make an uppercase version of `userName` with the `userName.upper()` method.**

2. **Assign the uppercase value back to `userName`.**

   The `userName.upper()` construct makes a new string that is the uppercase version of `userName`, but it doesn't directly change `userName`. If I really want `userName` to change (and I do in this case), I need to assign the converted value back to `userName` using the equal-sign assignment operation. Remember, most string methods don't change the original string. If you want to modify the string, you have to assign the results of the method back to the original variable.

3. **Replace all Rs with Ws.**

   The `replace()` method is a perfect tool for this. Since I converted the string to uppercase, I don't have to test for both uppercase and lowercase "r" because all "r"s are uppercase. Once again, I need to copy the value back to `userName`.

4. **Convert `userName` to title case. (Title case capitalizes the first letter of each word.)**

   I converted `username` to uppercase for convenience, but now `userName` needs to go back into a more readable format. Title case capitalizes only the first letter in the word, which is how names are usually formatted.

Slicing strings

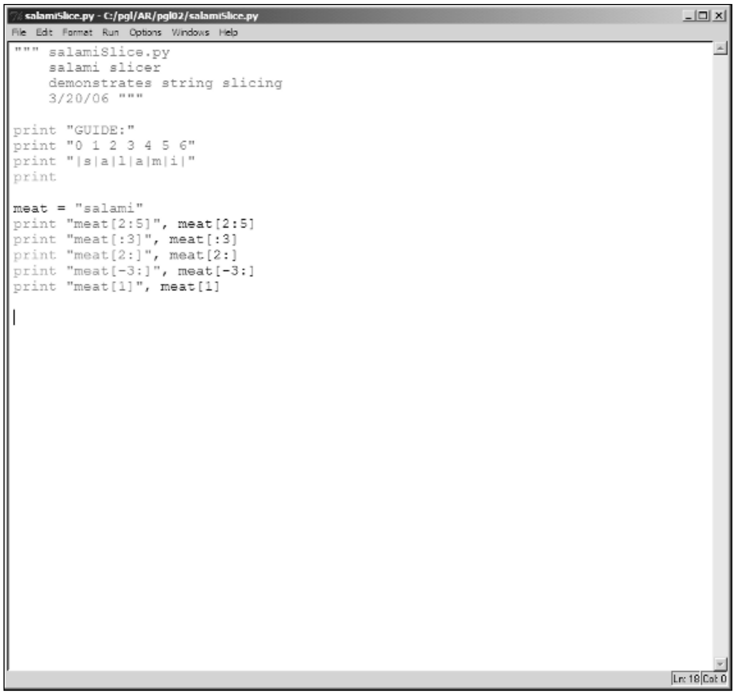
There's one more important thing to know about string variables (at least for now). You can cut them up to get new smaller strings. The Salami Slicer program (`salamiSlice.py`) illustrates how this works:

```
""" salamiSlice.py
    salami slicer
    demonstrates string slicing
    3/20/06 """

print "GUIDE:"
print "0 1 2 3 4 5 6"
print "|s|a|l|a|m|i|"
print

meat = "salami"
print "meat[2:5]", meat[2:5]
print "meat[:3]", meat[:3]
print "meat[2:]", meat[2:]
print "meat[-3:]", meat[-3:]
print "meat[1]", meat[1]
```

Figure 1-14 shows the output of the slicing program.


salamiSlice.py - C:/pgl/AR/pgl02/salamiSlice.py
File Edit Format Run Options Windows Help
Ln: 18 Col: 0

```
""" salamiSlice.py
    salami slicer
    demonstrates string slicing
    3/20/06 """

print "GUIDE:"
print "0 1 2 3 4 5 6"
print "|s|a|l|a|m|i|"
print

meat = "salami"
print "meat[2:5]", meat[2:5]
print "meat[:3]", meat[:3]
print "meat[2:]", meat[2:]
print "meat[-3:]", meat[-3:]
print "meat[1]", meat[1]
```

Figure 1-14: You can slice a string a lot of different ways.

Look carefully at the guide in Figure 1-14; it shows several positions in the string. In Python strings, character positions are best thought of as occurring *between* the characters. Slot 0 happens before the first character, 1 is between the first and second character, and so on. If you want to return a small part of a string, you can make a slice of the program, using square braces (`[]`) to indicate a starting point and an ending point. An example is a lot better than a lot of words, so imagine (as in the `salamiSlicer.py` program) you have a variable called `meat` that contains the value `"salami"`. Try this experiment to re-create the effect of the `salamiSlicer.py` program in your console:

1. **Create a `meat` variable.**

   Start by making `meat` contain the value `"salami"` — just type the following into the Python or IDLE console:

   ```
   >>> meat = "salami"
   ```

   Of course, you can make any variable you want, but to make sense of this mini-tutorial, go along with my examples first and then change them around later, just to make sure you know what's going on.

2. **Get a slice of salami.**

   Type `meat[2:5]` into the console to see how Python slices up the string.

   The program takes a slice of the value `"salami"` between slots 2 and 5. Look at the guide printed in Figure 1-14 to see that slot 2 occurs between the first "a" and the "l" in "salami." (Remember, these character slots occur *between* letters.) Slot 5 is the position between the "m" and the "i" in "salami." So the slice `meat[2:5]` returns the value `'lam'` when `meat` contains the value `'salami'`. If `meat` contained the value `'chicken'`, `meat[2:5]` would be `'ick'`.

3. **Get the first three characters.**

   The command `meat[:3]` uses a special trick in string slicing. If you leave out the first value, Python assumes you're starting at the beginning of the text, so `meat[:3]` is the same as `meat[0:3]`. When `meat` is "salami," `meat[:3]` is `'sal'`. You can also think of `meat[:3]` as the first three letters of `meat`.

4. **Get characters starting from any slot and going to the end of the word.**

   If you leave out the second value in the slicing operation, Python assumes you mean the end of the word. So, if you say `meat[2:]`, Python returns the values from slot 2 to the end of the word. If `meat` is "salami," `meat[2:]` means "take a slice from position 2 to the end of the word," and Python returns the value `'lami'`.

5. **Get the last few characters of a word.**

   You can use negative values in a slice; Python will count from the *end* of the string, rather than from the beginning. This is really handy when you want the last few letters of a string. The expression `meat[-3:]` means "take a slice that begins three characters from the end of the string and goes to the last character of the string." `meat[-3:]` will return `'ami'` if `meat` is "salami." Negative slots generally count from right to left.

6. **Get a specific character.**

   If you give only one value in the square braces, Python assumes you are talking about the letter immediately to the right of the given slot. `meat[4]` is correctly interpreted as "the character between slots 4 and 5."

Watch Your Step

Some people like to think of a single-value slice as indicating character position, but I think this is confusing to beginners. It isn't exactly correct to say that `meat[4]` means "the fourth character in the `meat` variable." If `meat` contains the value `'salami'`, then `meat[4]` is indeed "m" — but if I asked you to tell me the fourth character in the word "salami," you probably wouldn't say "m," because "m" is actually the *fifth* character in the word. The confusion stems from the fact that Python begins counting with zero. By Python's reasoning, "s" is not character 1, but character 0! It's much better to think of these character positions as the slots between letters.

Watch Your Step

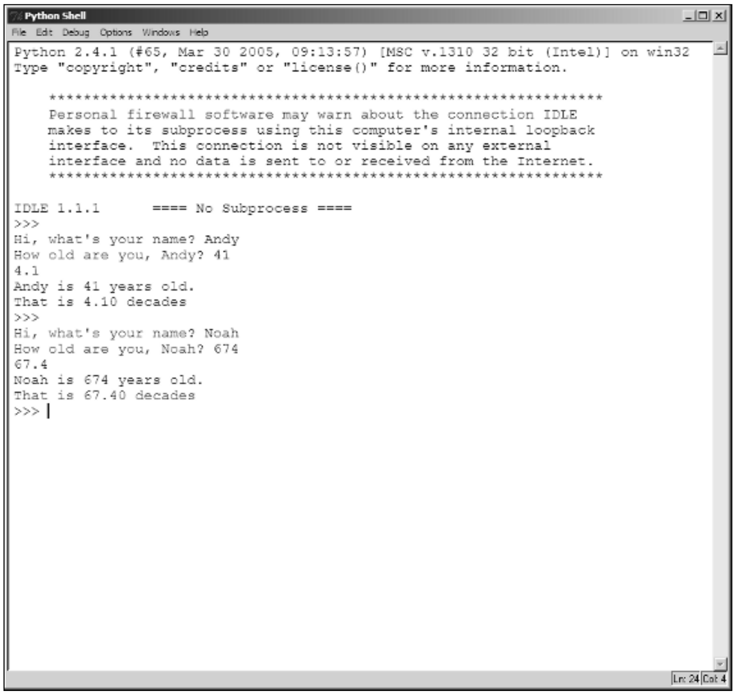
You might think you could indicate the end of the string using `-0`, but it doesn't work.

```
meat[-0] is 's'
```

This is because mathematically, there is no such thing as negative zero. Zero is zero, regardless of the sign you put in front of it. Just use nothing to indicate the end of the string, as I do in the example.

Interpolating text

Most of your programs so far have consisted of getting information from the user, manipulating that information, and sending it back to the user. Your output often consists of a mixture of text and variables, which can be confusing. It's even worse when you mix numeric and non-numeric values together, because you can't concatenate strings with numbers. Python offers an elegant solution called *variable interpolation*. (It's another one of those fancy words for an easy idea. Try to work it into your next social conversation.) Figure 1-15 shows a silly program that calculates a user's age in decades. I ran it a few times so you can see various outputs.



```
Python 2.4.1 (#65, Mar 30 2005, 09:13:57) [MSC v.1310 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.

    ****************************************************************
    Personal firewall software may warn about the connection IDLE
    makes to its subprocess using this computer's internal loopback
    interface.  This connection is not visible on any external
    interface and no data is sent to or received from the Internet.
    ****************************************************************

IDLE 1.1.1      ==== No Subprocess ====
>>> 
Hi, what's your name? Andy
How old are you, Andy? 41
4.1
Andy is 41 years old.
That is 4.10 decades
>>> 
Hi, what's your name? Noah
How old are you, Noah? 674
67.4
Noah is 674 years old.
That is 67.40 decades
>>> 
```

Figure 1-15: I don't actually know when you'd need your age in decades . . .

Read through the code, and you'll see how handy variable interpolation can be.

```
""" interpolation.py
    demonstrates variable interpolation
    """

name = raw_input("Hi, what's your name? ")

prompt = "How old are you, %s? " % name

age = raw_input(prompt)
age = int(age)

decades = age / 10.0
print decades

print "%s is %d years old." % (name, age)
print "That is %.2f decades." % decades
```

The program uses variable interpolation several different ways:

- **Build a prompt using the `name` variable.** The first example of variable interpolation comes in this line:

  ```
  prompt = "How old are you, %s? " % name
  ```

  Notice the special placeholder `%s`. This symbol means "put a string here." It allows me to continue writing the prompt without worrying about exactly what string value will go into the place held by the `%s` symbol. The next percent sign (`%`) indicates a value is coming up. Finally, `name` is the variable. When this line executes, Python builds a new string. When it sees the `%s` symbol, it looks for a string after the second `%` and places the value of that variable directly into the `prompt` string.

  Information Kiosk

  It isn't absolutely necessary to build the prompt in a separate string as I did here. I could have combined the string interpolation into the `raw_input` statement like this:

  ```
  age = raw_input("How old are you, %s? " % name)
  ```

  It's generally better to do only one job per line when you're starting out. When you get more comfortable with these ideas, you can combine them.

- **Interpolate two variables into one string.** The next interpolation statement is a little different:

  ```
  print "%s is %d years old." % (name, age)
  ```

The string inside has two symbols in it. The `%s` symbol represents a string. The `%d` indicates a decimal value (a base-10 integer) will be placed there. Since this string now expects *two* values (a string and a number), I feed it two variables. The `name` variable contains a string value and the `age` variable contains an integer.

Note that you must place the variables inside parentheses and separate them with commas if you have more than one. This type of structure is called a *tuple*. Tuples are values inside parentheses, separated by commas.

Watch Your Step

Make sure you supply enough values for your format string, or you will get an error.

Determine the formatting of floating-point values. One of the neatest things about variable interpolation is the control you have over the output. Floating-point numbers often cause problems because they can have many digits after the decimal point, but you may wish to display only a few. Look at how I solved this problem in the next line:

```
print "That is %.2f decades." % decades
```

The `decades` variable is a floating-point value.

Transfer

I divided `age` by the floating-point value `10.0` to produce a `float` response. Check Chapter 2 for more information on this process.

If you use `%f` in the format string, you are indicating that you will place a floating-point value in the output, but you don't indicate the length of that value. Usually you'll want to constrain the output to a reasonable length when you print floating-point values, because they can be difficult to read. By specifying `%.2f`, I am telling Python to print out a floating-point value with two places after the decimal point. The value is automatically rounded, using the standard algorithm.

Step Into the Real World

Many other programming languages provide some variation of variable interpolation, but they are all a little different. Python's version is really nice for a couple of reasons. First, it's pretty easy to use and understand. Also, it automatically converts values to strings and builds a nice string output. It also supports some advanced features when you use a more sophisticated data structure.

Street Jargon

argument: The part of a statement that Python will operate on. For example, in a `print` statement, the text that will actually be printed to the screen is the argument.

comment: A comment is a specially marked element in a program that is ignored by Python. Comments are useful for explaining what you're trying to accomplish.

compiled language: In compiled languages, the code is translated completely into a machine-readable format before it runs. In general, compiled languages provide faster-running programs, but interpreted languages can be more flexible.

console: A text-based interface. You can interact with Python at the DOS or UNIX shells in a console mode. When you use IDLE, you are also presented with a special console interface. The `print` command and `raw_input()` function are used to interact with the user through the console.

docstring: A special text value that occurs at the beginning of a Python program (and in some other places that will be described in later chapters). Docstrings are used to create the automatic documentation for the program, and they are usually a multi-line string. In this book, docstring values will all begin and end with the triple quote character.

IDLE (Interactive DeveLopment Environment): A basic-but-powerful text editor and Python environment that ships with most versions of Python. Note that Eric Idle was a member of the Monty Python comedy troupe. I don't know whether there is a CLEESE or GILLIAM editor, but there should be.

interactive mode: A special way of interacting directly with the Python environment. When you use the interactive mode, you type commands directly into Python rather than storing them in a text file. Interactive mode is useful for quick code checks and accessing the online help system.

interpreted language: An interpreted language is one translated from a human-like dialect into machine-readable code in real time. Programs written in an interpreted language cannot run unless the user also has the interpreter program installed on the same system. Interpreted languages can be slow, but the programmer can interact with them in real time. The primary alternative to an interpreted language is a compiled language (such as C and C++).

method: Something an object knows how to do. For example, Python's built-in string object has a method to convert to uppercase (`string.upper()`).

object-oriented programming (OOP): A form of programming that allows the programmer to organize code in objects much like those in the real world. Most built-in Python elements (lists, variables, and so on) are objects.

Street Jargon *continued*

pygame: A module that adds game-programming functionality to Python. The pygame library will be introduced more thoroughly beginning in Chapter 4.

Python: An interpreted programming language designed to be powerful yet easy for beginners.

SDL (Simple DirectMedia Layer): A reasonably universal graphics package. The pygame library is a special Python adaptation of the SDL graphics library.

slicing: Using the square brace (`[]`) operators to extract a subset of a string or list.

statement: A command or line of code in a programming language.

string: A variable containing text.

string interpolation: A technique for embedding variables into a string for easy formatting.

syntax: A programming language's rules of grammar and punctuation.

tuple: Values inside parentheses, separated by commas. The elements of a tuple cannot be changed.

variable: A variable is a reference to a specific place in a computer's memory meant to store information. Each variable has a name and a value.

TOKENS

Command	Arguments	Description	Example
`print` *`value`*	*`value`*: what will be printed to the screen	Prints something to the console. A value can be text in quotes or a variable name.	`print "Hi there"`
`variable` `= raw_input(`*`prompt`*`)`	*`variable`*: what receives the entered value, in this case *`prompt`* (a question to ask the user)	Prompts for a user response (usually text encased in quotes) and stores the response in a variable.	`username = raw_input("What is your name")`

Last Stop

Practice Exam

1. **Describe three ways game programming is useful for beginning programmers.**

2. **Describe three reasons to use Python as a first game programming language.**

3. **Which is the best description of a variable in Python?**

A) A place to hold numbers but not text data.

B) A named place in memory to hold information.

C) A command without an argument.

D) Python doesn't have variables. It uses constants instead.

4. **Which of the following is the best name for a variable to hold a person's shoe size?**

A) `ss`

B) `thePersonsShoeSize`

C) `7 1/2 wide`

D) `shoeSize`

5. **What's the best way to read the code `result = 5 + 7`?**

6. **Why might you use IDLE for writing your Python programs?**

7. **Why should your programs begin with a docstring?**

8. **Why are comments used in Python?**

9. **How does the `print` statement work?**

10. **True or false (and explain why): The `print` statement prints quoted text differently than arguments without quotes.**

11. **If the `raw_input()` function is used for input, why does it output something to the screen?**

12. **Write a program that asks the user's name and then responds with a customized greeting. For example, if the user's name is "Elizabeth," the program says `"Hi, Elizabeth!"`**

____

____

____

13. **What is an object method?**

A) Something that happens to an object

B) A characteristic of an object

C) An attribute

D) Something an object can do

14. **Name some methods of the string object.**

____

____

____

15. **Describe the result of the following code:**

```
var = "programming"
print var[3:7]
```

A) `'gram'`

B) `'prog'`

C) `'programming'`

D) `'GRAM'`

16. **What is the primary purpose of string slicing?**

____

____

____

EXIT