Chapter 3

Interacting with Python

In This Chapter

- Accessing the command line
- Using commands to perform tasks
- Obtaining help about Python
- Ending a command-line session

Ultimately, any application you create interacts with the computer and the data it contains. The focus is on data because without data, there isn't a good reason to have an application. Any application you use (even one as simple as Solitaire) manipulates data in some way. In fact, the acronym CRUD sums up what most applications do:

- Create
- Read
- Update
- Delete

If you remember CRUD, you'll be able to summarize what most applications do with the data your computer contains (and some applications really are quite cruddy). However, before your application accesses the computer, you have to interact with a programming language that creates a list of tasks to perform in a language the computer understands. That's the purpose of this chapter. You begin interacting with Python. Python takes the list of steps you want to perform on the computer's data and changes those steps into bits the computer understands.

Understanding the importance of the README file

Many applications include a README file. The README file usually provides updated information that didn't make it into the documentation before the application was put into a production status. Unfortunately, most people ignore the README file and some don't even know it exists. As a result, people who should know something interesting about their shiny new product never find out. Python has a `README.txt` file in the `\Python33` directory. When you open this file, you find all sorts of really interesting information:

- How to build a copy of Python for Linux systems
- Where to find out about new features in this version of Python
- Where to find the latest version of the Python documentation
- How to convert your older Python applications to work with Python 3.3.*x*
- What you need to do to test custom Python modifications
- How to install multiple versions of Python on the same system
- How to access bug and issue tracking for Python
- How to request updates to Python
- How to find out when the next version of Python will come out

Opening and reading the README file will help you become a Python genius. People will be amazed that you really do know something interesting about Python and will ask you all sorts of questions (deferring to your wisdom). Of course, you could always just sit there, thinking that the README is just too much effort to read.

Opening the Command Line

Python offers a number of ways to interact with the underlying language. For example, you worked a bit with the Integrated DeveLopment Environment (IDLE) in Chapter 2. IDLE makes it easy to develop full-fledged applications. However, sometimes you simply want to experiment or to run an existing application. Often, using the command-line version of Python works better in these cases because it offers better control over the Python environment through command-line switches, uses fewer resources, and relies on a minimalistic interface so that you can focus on trying out code rather than playing with a GUI.

Starting Python

Depending on your platform, you might have multiple ways to start the command line. Here are the methods that are commonly available:

- Select the Python (command-line) option found in the `Python 3.3` folder. This option starts a command-line session that uses the default settings.
- Open a command prompt or terminal, type **Python**, and press Enter. Use this option when you want greater flexibility in configuring the Python environment using command-line switches.
- Locate the Python folder, such as `C:\Python33` in Windows, and open the `Python.exe` file directly. This option also opens a command-line session that uses the default settings, but you can do things like open it with increased privileges (for applications that require access to secured resources) or modify the executable file properties (to add command-line switches).

No matter how you start Python at the command line, you eventually end up with a prompt similar to the one shown in Figure 3-1. (Your screen may look slightly different from the one shown in Figure 3-1 if you rely on a platform other than Windows, you're using IDLE instead of the command-line version of Python, your system is configured differently from mine, or you have a different version of Python.) This prompt tells you the Python version, the host operating system, and how to obtain additional information.

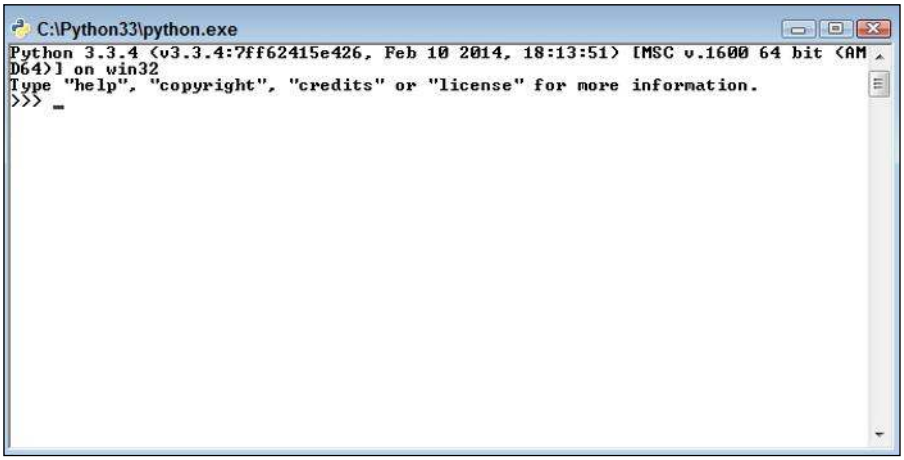

Figure 3-1: The Python command prompt tells you a bit about the Python environment.

Using the command line to your advantage

This section will seem a little complicated at first, and you won't normally need this information when using the book. However, it's still good information, and you'll eventually need it. For now, you can browse the information so that you know what's available and then come back to it later when you really do need the information.

To start Python at a command prompt, type **Python** and press Enter. However, that's not all you can do. You can also provide some additional information to change how Python works:

- **Options:** An option, or command-line switch, begins with a minus sign followed by one or more letters. For example, if you want to obtain help about Python, you type **Python –h** and press Enter. You see additional information about how to work with Python at the command line. The options are described later in this section.
- **Filename:** Providing a filename as input tells Python to load that file and run it. You can run any of the example applications from the downloadable code by providing the name of the file containing the example as input. For example, say that you have an example named `SayHello.py`. To run this example, you type **Python SayHello.py** and press Enter.
- **Arguments:** An application can accept additional information as input to control how it runs. This additional information is called an argument. Don't worry too much about arguments right now — they appear later in the book.

Most of the options won't make sense right now. They're here so that you can find them later when you need them (this is the most logical place to include them in the book). Reading through them will help you gain an understanding of what's available, but you can also skip this material until you need it later.

Python uses case-sensitive options. For example, `-s` is a completely different option from `-S`. The Python options are

- `-b`: Add warnings to the output when your application uses certain Python features that include: `str(bytes_instance)`, `str(bytearray_instance)`, and comparing `bytes` or `bytearray` with `str()`.
- `-bb`: Add errors to the output when your application uses certain Python features that include: `str(bytes_instance)`, `str(bytearray_instance)`, and comparing `bytes` or `bytearray` with `str()`.
- `-B`: Don't write `.py` or `.pyco` files when performing a module import.

- `-c cmd`: Use the information provided by *cmd* to start a program. This option also tells Python to stop processing the rest of the information as options (it's treated as part of the command).
- `-d`: Start the debugger (used to locate errors in your application).
- `-E`: Ignore all the Python environment variables, such as PYTHONPATH, that are used to configure Python for use.
- `-h`: Display help about the options and basic environment variables onscreen. Python always exits after it performs this task without doing anything else so that you can see the help information.
- `-i`: Force Python to let you inspect the code interactively after running a script. It forces a prompt even if `stdin` (the standard input device) doesn't appear to be a terminal.
- `-m mod`: Run the library module specified by *mod* as a script. This option also tells Python to stop processing the rest of the information as options (the rest of the information is treated as part of the command).
- `-O`: Optimize the generated bytecode slightly (makes it run faster).
- `-OO`: Perform additional optimization by removing doc-strings.
- `-q`: Tell Python not to print the version and copyright messages on interactive startup.
- `-s`: Force Python not to add the user site directory to `sys.path` (a variable that tells Python where to find modules).
- `-S`: Don't run `'import site'` on initialization. Using this option means that Python won't look for paths that may contain modules it needs.
- `-u`: Allow unbuffered binary input for the `stdout` (standard output) and `stderr` (standard error) devices. The `stdin` device is always buffered.
- `-v`: Place Python in verbose mode so that you can see all the import statements. Using this option multiple times increases the level of verbosity.
- `-V`: Display the Python version number and exit.
- `--version`: Display the Python version number and exit.
- `-W arg`: Modify the warning level so that Python displays more or fewer warnings. The valid *arg* values are
  - `action`
  - `message`
  - `category`
  - `module`
  - `lineno`

- `-x`: Skip the first line of a source code file, which allows the use of non-Unix forms of `#!cmd`.
- `-X opt`: Set an implementation-specific option. (The documentation for your version of Python discusses these options, if there are any.)

Using Python environment variables to your advantage

Environment variables are special settings that are part of the command line or terminal environment for your operating system. They serve to configure Python in a consistent manner. Environment variables perform many of the same tasks as do the options that you supply when you start Python, but you can make environment variables permanent so that you can configure Python the same way every time you start it without having to manually supply the option.

As with options, most of these environment variables won't make any sense right now. You can read through them to see what is available. You find some of the environment variables used later in the book. Feel free to skip the rest of this section and come back to it later when you need it.

Most operating systems provide the means to set environment variables temporarily, by configuring them during a particular session, or permanently, by configuring them as part of the operating system setup. Precisely how you perform this task depends on the operating system. For example, when working with Windows, you can use the `Set` command (see my blog post at `http://blog.johnmuellerbooks.com/2014/02/24/using-the-set-command-to-your-advantage/` for details) or rely on a special Windows configuration feature (see my post at `http://blog.johnmuellerbooks.com/2014/02/17/adding-a-location-to-the-windows-path/` for setting the `Path` environment variable as an example).

Using environment variables makes sense when you need to configure Python the same way on a regular basis. The following list describes the Python environment variables:

- `PYTHONCASEOK=x`: Forces Python to ignore case when parsing `import` statements. This is a Windows-only environment variable.
- `PYTHONDEBUG=x`: Performs the same task as the `-d` option.
- `PYTHONDONTWRITEBYTECODE=x`: Performs the same task as the `-B` option.

- `PYTHONFAULTHANDLER=x`: Forces Python to dump the Python traceback (list of calls that led to an error) on fatal errors.
- `PYTHONHASHSEED=arg`: Determines the seed value used to generate hash values from various kinds of data. When this variable is set to `random`, Python uses a random value to seed the hashes of `str`, `bytes`, and `datetime` objects. The valid integer range is 0 to 4294967295. Use a specific seed value to obtain predictable hash values for testing purposes.
- `PYTHONHOME=arg`: Defines the default search path that Python uses to look for modules.
- `PYTHONINSPECT=x`: Performs the same task as the `-i` option.
- `PYTHONIOENCODING=arg`: Specifies the `encoding[:errors]` (such as `utf-8`) used for the `stdin`, `stdout`, and `stderr` devices.
- `PYTHONNOUSERSITE`: Performs the same task as the `-s` option.
- `PYTHONOPTIMIZE=x`: Performs the same task as the `-O` option.
- `PYTHONPATH=arg`: Provides a semicolon (;) separated list of directories to search for modules. This value is stored in the `sys.path` variable in Python.
- `PYTHONSTARTUP=arg`: Defines the name of a file to execute when Python starts. There is no default value for this environment variable.
- `PYTHONUNBUFFERED=x`: Performs the same task as the `-u` option.
- `PYTHONVERBOSE=x`: Performs the same task as the `-v` option.
- `PYTHONWARNINGS=arg`: Performs the same task as the `-W` option.

Typing a Command

After you start the command-line version of Python, you can begin typing commands. Using commands makes it possible to perform tasks, test ideas that you have for writing your application, and discover more about Python. Using the command line lets you gain hands-on experience with how Python actually works — details that could be hidden by an Interactive Development Environment (IDE) such as IDLE. The following sections get you started using the command line.

Telling the computer what to do

Python, like every other programming language in existence, relies on commands. A *command* is simply a step in a procedure. In Chapter 1, you saw how "Get the bread and butter from the refrigerator" is a step in a procedure for making toast. When working with Python, a command, such as `print()`, is simply the same thing: a step in a procedure.

To tell the computer what to do, you issue one or more commands that Python understands. Python translates these commands into instructions that the computer understands, and then you see the result. A command such as `print()` can display the results onscreen so that you get an instant result. However, Python supports all sorts of commands, many of which don't display any results onscreen but still do something important.

As the book progresses, you use commands to perform all sorts of tasks. Each of these tasks will help you accomplish a goal, just as the steps in a procedure do. When it seems as if all the Python commands become far too complex, simply remember to look at them as steps in a procedure. Even human procedures become complex at times, but if you take them one step at a time, you begin to see how they work. Python commands are the same way. Don't get overwhelmed by them; instead, look at them one at a time and focus on just that step in your procedure.

Telling the computer you're done

At some point, the procedure you create ends. When you make toast, the procedure ends when you finish buttering the toast. Computer procedures work precisely the same way. They have a starting and an ending point. When typing commands, the ending point for a particular step is the Enter key. You press Enter to tell the computer that you're done typing the command. As the book progresses, you find that Python provides a number of ways to signify that a step, group of steps, or even an entire application is complete. No matter how the task is accomplished, computer programs always have a distinct starting and stopping point.

Seeing the result

You now know that a command is a step in a procedure and that each command has a distinct starting and ending point. In addition, groups of commands and entire applications also have a distinct starting and ending

point. So, take a look at how this works. The following procedure helps you see the result of using a command:

1. **Start a copy of the Python command-line version.**

   You see a command prompt where you can type commands, as shown in Figure 3-1.

2. **Type** print("This is a line of text.") **at the command line.**

   Notice that nothing happens. Yes, you typed a command, but you haven't signified that the command is complete.

3. **Press Enter.**

   The command is complete, so you see a result like the one shown in Figure 3-2.

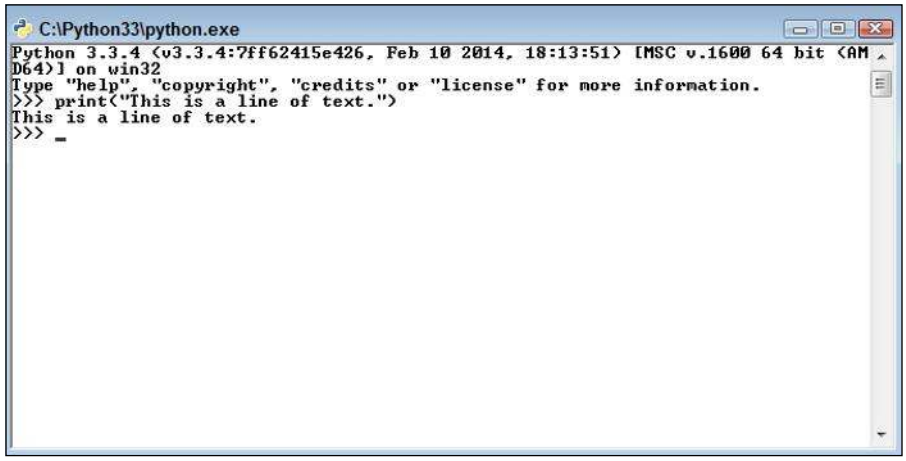

Figure 3-2: Issuing commands tells Python what to tell the computer to do.

This exercise shows you how things work within Python. Each command that you type performs some task, but only after you tell Python that the command is complete in some way. The `print()` command displays data onscreen. In this case, you supplied text to display. Notice that the output shown in Figure 3-2 comes immediately after the command because this is an interactive environment — one in which you see the result of any given command immediately after Python performs it. Later, as you start creating applications, you notice that sometimes a result doesn't appear immediately because the application environment delays it. Even so, the command is executed by Python immediately after the application tells Python that the command is complete.

Using Help

Python is a computer language, not a human language. As a result, you won't speak it fluently at first. If you think about it for a moment, it makes sense that you won't speak Python fluently (and as with most human languages, you won't know every command even after you do become fluent). Having to discover Python commands a little at a time is the same thing that happens when you learn to speak another human language. If you normally speak English and try to say something in German, you find that you must have some sort of guide to help you along. Otherwise, anything you say is gibberish and people will look at you quite oddly. Even if you manage to say something that makes sense, it may not be what you want. You might go to a restaurant and order hot hubcaps for dinner when what you really wanted was a steak.

Likewise, when you try to speak Python, you need a guide to help you. Fortunately, Python is quite accommodating and provides immediate help to keep you from ordering something you really don't want. The help provided inside Python works at two levels:

- **Help mode,** in which you can browse the available commands
- **Direct help,** in which you ask about a specific command

There isn't a correct way to use help — just the method that works best for you at a particular time. The following sections describe how to obtain help.

Getting into help mode

When you first start Python, you see a display similar to the one shown in Figure 3-1. Notice that Python provides you with four commands at the outset (which is actually your first piece of help information):

- `help`
- `copyright`
- `credits`
- `license`

All four commands provide you with help, of a sort, about Python. For example, the `copyright()` command tells you about who holds the right to copy, license, or otherwise distribute Python. The `credits()` command tells you

who put Python together. The `license()` command describes the usage agreement between you and the copyright holder. However, the command you most want to know about is simply `help()`.

To enter help mode, type **help()** and press Enter. Notice that you must include the parentheses after the command even though they don't appear in the help text. Every Python command has parentheses associated with it. After you enter this command, Python goes into help mode and you see a display similar to the one shown in Figure 3-3.

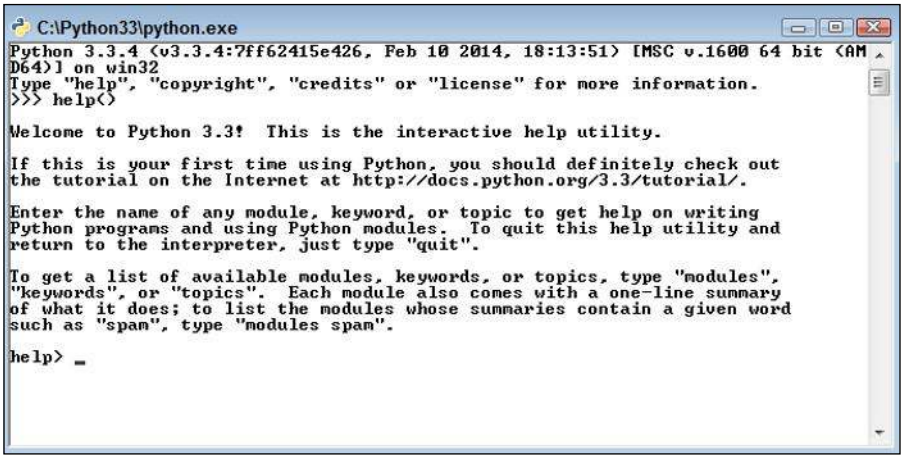

Figure 3-3: You ask Python about other commands in help mode.

You can always tell that you're in help mode by the `help>` prompt that you see in the Python window. As long as you see the `help>` prompt, you know that you're in help mode.

Asking for help

To obtain help, you need to know what question to ask. The initial help message that you see when you go into help mode (refer to Figure 3-3) provides some helpful tips about the kinds of questions you can ask. If you want to explore Python, the three basic topics are

- `modules`
- `keywords`
- `topics`

The first two topics won't tell you much for now. You won't need the `modules` topic until Chapter 10. The `keywords` topic will begin proving useful in Chapter 4. However, the `topics` keyword is already useful because it helps you understand where to begin your Python adventure. To see what topics are available, type **topics** and press Enter. You see a list of topics similar to those shown in Figure 3-4.

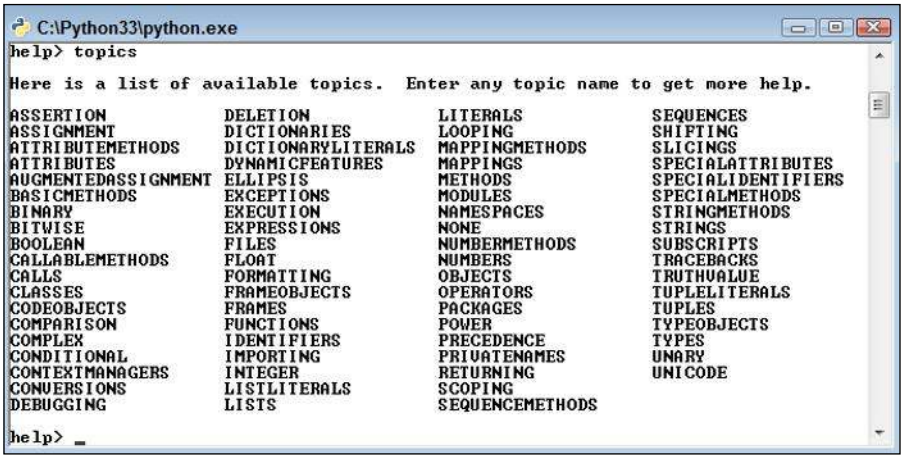

Figure 3-4: The `topics` help topic provides you with a starting point for your Python adventure.

When you see a topic that you like, such as `FUNCTIONS`, simply type that topic and press Enter. To see how this works, type **FUNCTIONS** and press Enter (you must type the word in uppercase — don't worry, Python won't think you're shouting). You see help information similar to that shown in Figure 3-5.

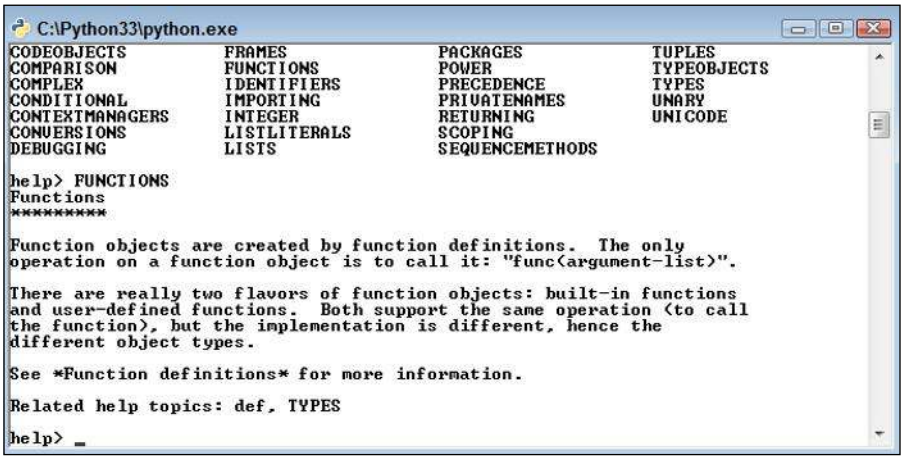

Figure 3-5: You must use uppercase when requesting topic information.

As you work through examples in the book, you use commands that look interesting, and you might want more information about them. For example, in the "Seeing the result" section of this chapter, you use the `print()` command. To see more information about the `print()` command, type **print** and press Enter (notice that you don't include the parentheses this time because you're requesting help about `print()`, not actually using the command). Figure 3-6 shows typical help information for the `print()` command.

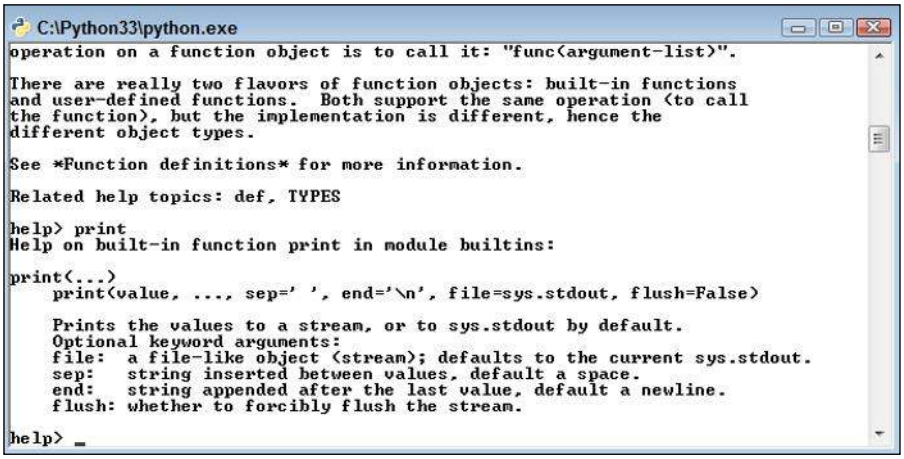

Figure 3-6: Request command help information by typing the command using whatever case it actually uses.

Unfortunately, reading the help information probably doesn't help much yet because you need to know more about Python. However, you can ask for more information. For example, you might wonder what `sys.stdout` means — and the help topic certainly doesn't tell you anything about it. Type **sys.stdout** and press Enter. You see the help information shown in Figure 3-7.

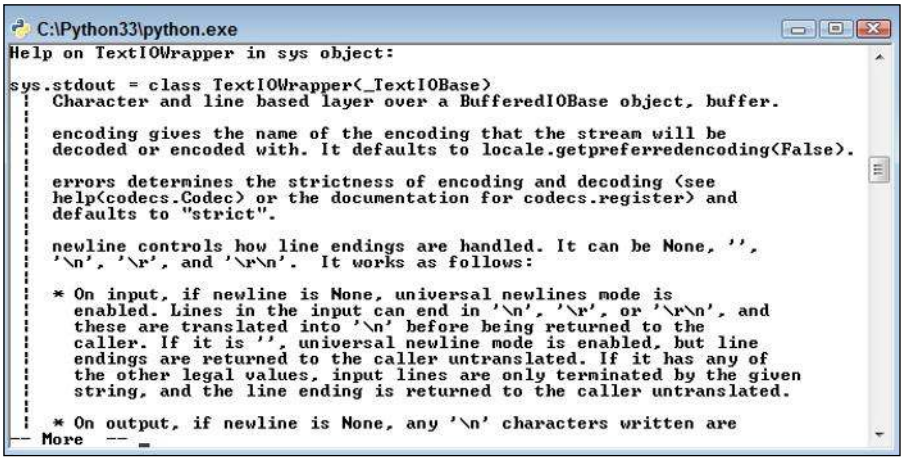

Figure 3-7: You can ask for help on the help you receive.

You may still not find the information as helpful as you need, but at least you know a little more. In this case, help has a lot to say and it can't all fit on one screen. Notice the following entry at the bottom of the screen:

```
-- More  --
```

To see the additional information, press the spacebar. The next page of help appears. As you read to the bottom of each page of help, you can press the spacebar to see the next page. The pages don't go away — you can scroll up to see previous material.

Leaving help mode

At some point, you need to leave help mode to perform useful work. All you have to do is press Enter without typing anything. When you press Enter, you see a message about leaving help, and then the prompt changes to the standard Python prompt, as shown in Figure 3-8.

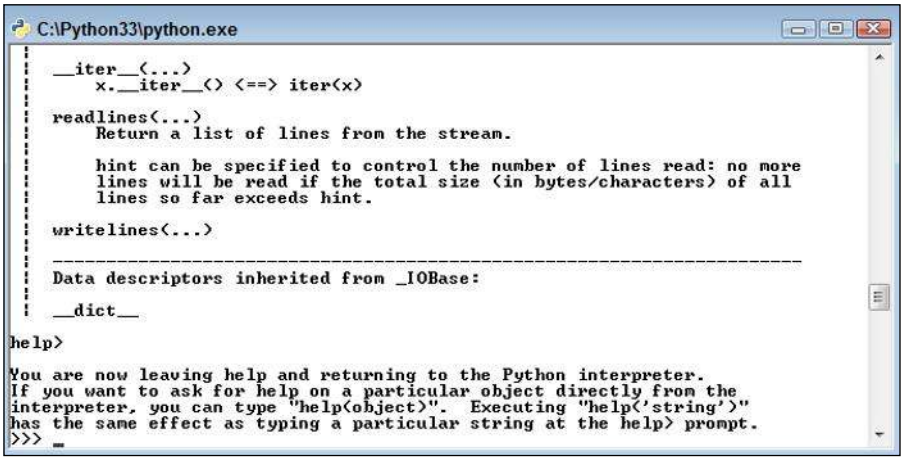

Figure 3-8: Exit help mode by pressing Enter without typing anything.

Obtaining help directly

Entering help mode isn't necessary unless you want to browse, which is always a good idea, or unless you don't actually know what you need to find. If you have a good idea of what you need, all you need to do is ask for help directly (a really nice thing for Python to do). So, instead of fiddling with help mode, you simply type the word *help,* followed by a left parenthesis and

single quote, whatever you want to find, another single quote, and the right parenthesis. For example, if you want to know more about the `print()` command, you type **help('print')** and press Enter. Figure 3-9 shows typical output when you access help this way.

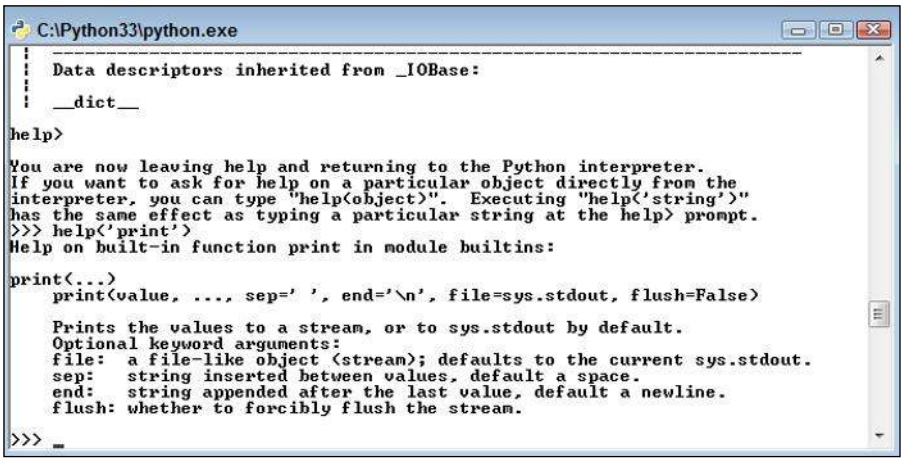

Figure 3-9: Python makes it possible to obtain help whenever you need it without leaving the Python prompt.

You can browse at the Python prompt, too. For example, when you type **help('topics')** and press Enter, you see a list of topics like the one that appears in Figure 3-10. You can compare this list with the one shown in Figure 3-4. The two lists are identical, even though you typed one while in help mode and the other while at the Python prompt.

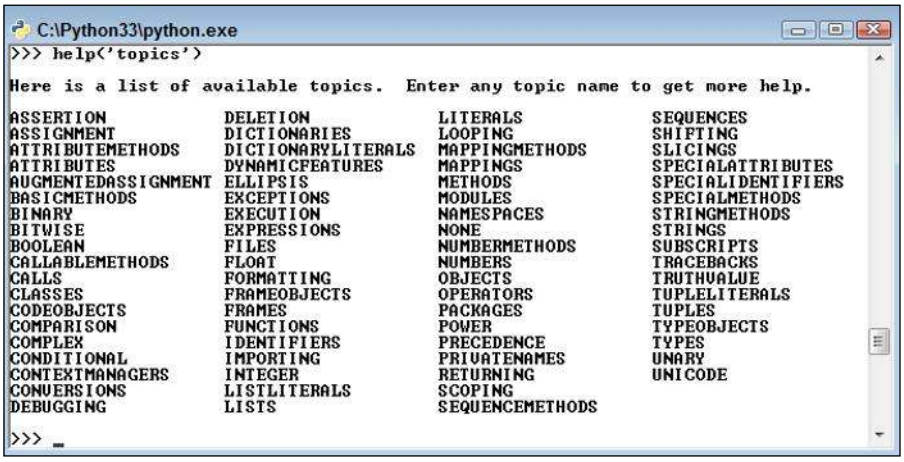

Figure 3-10: It's possible to browse at the Python prompt if you really want to.

You might wonder why Python has a help mode at all if you can get the same results at the Python prompt. The answer is convenience. It's easier to browse in the help mode. In addition, even though you don't do a lot of extra typing at the prompt, you do perform less typing while in help mode. Help mode also provides additional helps, such as by listing commands that you can type, as shown in Figure 3-3. So you have all kinds of good reasons to enter help mode when you plan to ask Python a lot of help questions.

No matter where you ask for help, you need to observe the correct capitalization of help topics. For example, if you want general information about functions, you must type **help('FUNCTIONS')** and not `help('Functions')` or `help('functions')`. When you use the wrong capitalization, Python will tell you that it doesn't know what you mean or that it couldn't find the help topic. It won't know to tell you that you used the wrong capitalization. Someday computers will know what you meant to type, rather than what you did type, but that hasn't happened yet.

Closing the Command Line

Eventually, you want to leave Python. Yes, it's hard to believe, but people have other things to do besides playing with Python all day long. You have two standard methods for leaving Python and a whole bunch of nonstandard methods. Generally, you want to use one of the standard methods to ensure that Python behaves as you expect it to, but the nonstandard methods work just fine when you simply want to play around with Python and not perform any productive work. The two standard methods are

- `quit()`
- `exit()`

Either of these methods will close the interactive version of Python. The *shell* (the Python program) is designed to allow either command.

Both of these commands can accept an optional argument. For example, you can type **quit(5)** or **exit(5)** and press Enter to exit the shell. The numeric argument sets the command prompt's `ERRORLEVEL` environment variable, which you can then intercept at the command line or as part of a batch file. Standard practice is to simply use `quit()` or `exit()` when nothing has gone wrong with the application. To see this way of exiting at work, you must

1. **Open a command prompt or terminal.**

   You see a prompt.

2. **Type** Python **and press Enter to start Python.**

   You see the Python prompt.

3. **Type** quit(5) **and press Enter.**

   You see the prompt again.

4. **Type** echo %ERRORLEVEL% **and press Enter.**

   You see the error code, as shown in Figure 3-11. When working with platforms other than Windows, you may need to type something other than `echo %ERRORLEVEL%`. For example, when working with a bash script, you type **echo $** instead.

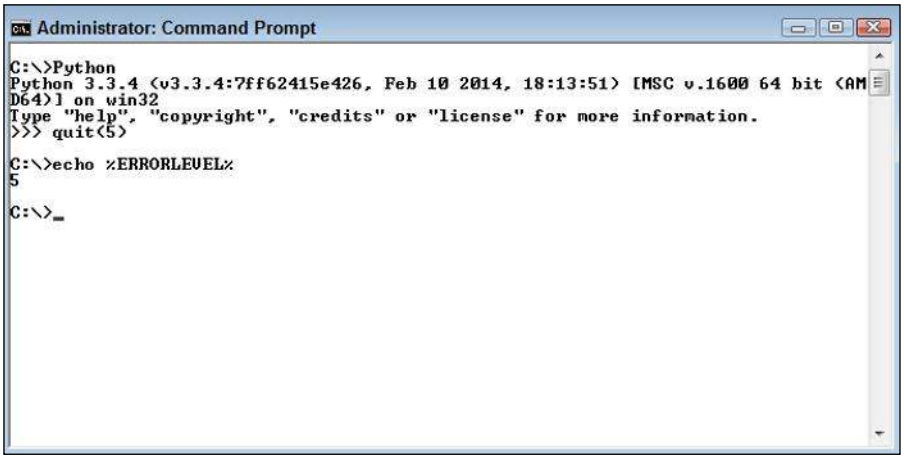

Figure 3-11: Add an error code when needed to tell others the Python exit status.

One of the most common nonstandard exit methods is to simply click the command prompt's or terminal's Close button. Using this approach means that your application may not have time to perform any required cleanup, which can result in odd behaviors. It's always better to close Python using an expected approach if you've been doing anything more than simply browsing.

You also have access to a number of other commands for closing the command prompt when needed. In most cases, you won't need these special commands, so you can skip the rest of this section if desired.

When you use `quit()` or `exit()`, Python performs a number of tasks to ensure that everything is neat and tidy before the session ends. If you suspect that a session might not end properly anyway, you can always rely on one of these two commands to close the command prompt:

- `sys.exit()`
- `os._exit()`

Both of these commands are used in emergency situations only. The first, `sys.exit()`, provides special error-handling features that you discover in Chapter 9. The second, `os._exit()`, exits Python without performing any of the usual cleanup tasks. In both cases, you must import the required module, either `sys` or `os`, before you can use the associated command. Consequently, to use the `sys.exit()` command, you actually use this code:

```
import sys
sys.exit()
```

You must provide an error code when using `os._exit()` because this command is used only when an extreme error has occurred. The call to this command will fail if you don't provide an error code. To use the `os._exit()` command, you actually use this code (where the error code is 5):

```
import os
os._exit(5)
```

Chapter 10 discusses importing modules in detail. For now, just know that these two commands are for special uses only and you won't normally use them in an application.