Learning\_Python\_3E\_Mark\_C03

34

Chapter 3CHAPTER 3

How You Run Programs 3

OK, it’s time to start running some code. Now that you have a handle on program execution, you’re finally ready to start some real Python programming. At this point, I’ll assume that you have Python installed on your computer; if not, see the prior chapter and Appendix A for installation and configuration hints. There are a variety of ways to tell Python to execute the code you type. This chapter discusses all the program-launching techniques in common use today. Along the way, you’ll learn how to type code interactively, and how to save it in files to be run with system command lines, icon clicks, module imports, IDE GUIs such as IDLE and Eclipse, and more.

Bien, es hora de comenzar a ejecutar código. Ahora que domina la ejecución del programa, finalmente está listo para comenzar a programar Python de verdad. En este punto, asumiré que tiene Python instalado en su computadora; si no, consulte el capítulo anterior y el Apéndice A para obtener sugerencias de instalación y configuración. Hay una variedad de formas de decirle a Python que ejecute el código que escribe. En este capítulo se analizan todas las técnicas de lanzamiento de programas de uso común en la actualidad. En el camino, aprenderá cómo escribir código de forma interactiva y cómo guardarlo en archivos para ejecutarlos con líneas de comando del sistema, clics en íconos, importaciones de módulos, GUI IDE como IDLE y Eclipse, y más.

If you just want to find out how to run a Python program quickly, you may be tempted to read the parts that pertain only to your platform and move on to Chapter 4. But don’t skip the material on module imports, as that’s essential to understanding Python’s program architecture. I also encourage you at least to skim the sections on IDLE and other IDEs, so you’ll know what tools are available for when you start developing more sophisticated Python programs.

Si solo desea saber cómo ejecutar un programa de Python rápidamente, puede sentirse tentado a leer las partes que pertenecen solo a su plataforma y continuar con el Capítulo 4. Pero no se salte el material sobre las importaciones de módulos, ya que eso es esencial. para comprender la arquitectura del programa de Python. También lo animo a que al menos hojee las secciones sobre IDLE y otros IDE, para que sepa qué herramientas están disponibles cuando comience a desarrollar programas de Python más sofisticados.

**Interactive Coding**

Perhaps the simplest way to run Python programs is to type them at Python’s interactive command line. There are a variety of ways to start this command line—in an IDE, from a system console, and so on. Assuming the interpreter is installed as an executable program on your system, the most platform-neutral way to start an interactive interpreter session is usually just to type python at your operating system’s prompt, without any arguments. For example:

Quizás la forma más sencilla de ejecutar programas de Python es escribirlos en la línea de comandos interactiva de Python. Hay varias formas de iniciar esta línea de comandos: en un IDE, desde una consola del sistema, etc. Suponiendo que el intérprete esté instalado como un programa ejecutable en su sistema, la forma más neutral de plataforma para iniciar una sesión de intérprete interactivo suele ser simplemente escribir python en el indicador de su sistema operativo, sin ningún argumento. Por ejemplo:

% python

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.

>>>

Typing the word “python” at your system shell prompt begins an interactive Python session (the “%” character stands for your system’s prompt, not input that you type yourself). The notion of a system shell prompt is generic, but exactly how you access the prompt varies by platform:

Al escribir la palabra "python" en el indicador de shell de su sistema, se inicia una sesión interactiva de Python (el carácter "%" representa el indicador de su sistema, no una entrada que escriba usted mismo). La noción de un indicador de shell del sistema es genérica, pero la forma exacta de acceder al indicador varía según la plataforma:

• On Windows, you can type python in a DOS console window (a.k.a. the Command Prompt, usually found in the Accessories section of the Programs menu of your Start button) or in the Start ➝ Run... dialog box.

• On Unix, Linux, and Mac OS X, you might type this command in a shell or terminal window (e.g., in an xterm or console running a shell such as ksh or csh).

• Other systems may use similar or platform-specific devices. On PalmPilots, for example, click the Python home icon to launch an interactive session.

If you have not set your shell’s PATH environment variable to include Python’s install directory, you may need to replace the word “python” with the full path to the Python executable on your machine. On Windows, try typing C:\Python25\python (for version 2.5); on Unix and Linux, /usr/local/bin/python (or /usr/bin/python) will often suffice. Alternatively, you can run a change-directory command to go to Python’s install directory before typing “python” (try a cd c:\python25 command on Windows, for instance).

Si no ha configurado la variable de entorno PATH de su shell para incluir el directorio de instalación de Python, es posible que deba reemplazar la palabra "python" con la ruta completa al ejecutable de Python en su máquina. En Windows, intente escribir C:\Python25\python (para la versión 2.5); en Unix y Linux, /usr/local/bin/python (o /usr/bin/python) suele ser suficiente. Alternativamente, puede ejecutar un comando de cambio de directorio para ir al directorio de instalación de Python antes de escribir "python" (pruebe con un comando cd c:\python25 en Windows, por ejemplo).

The Python interactive session begins by printing two lines of informational text (which I’ll omit from this book’s examples to save space), then prompts for input with >>> when it’s waiting for you to type a new Python statement or expression. When working interactively, the results of your code are displayed after the >>> lines. Here are the results of two Python print statements:

La sesión interactiva de Python comienza con la impresión de dos líneas de texto informativo (que omitiré de los ejemplos de este libro para ahorrar espacio), luego solicita la entrada con >>> cuando está esperando que escriba una nueva declaración o expresión de Python. Cuando se trabaja de forma interactiva, los resultados de su código se muestran después de las líneas >>>. Estos son los resultados de dos declaraciones de impresión de Python:

% python

>>> print 'Hello world!'

Hello world!

>>> print 2 \*\* 8

256

Again, you don’t need to worry about the details of the print statements shown here yet (we’ll start digging into syntax in the next chapter). In short, they print a Python string and an integer, as shown by the output lines that appear after each >>> input line. When working interactively like this, you can type as many Python commands as you like; each is run immediately after it’s entered. Moreover, because the interactive session automatically prints the results of expressions you type, you don’t usually need to say “print” explicitly at this prompt:

Nuevamente, aún no necesita preocuparse por los detalles de las instrucciones de impresión que se muestran aquí (comenzaremos a profundizar en la sintaxis en el próximo capítulo). En resumen, imprimen una cadena de Python y un número entero, como se muestra en las líneas de salida que aparecen después de cada línea de entrada >>>. Cuando trabaja de forma interactiva de esta manera, puede escribir tantos comandos de Python como desee; cada uno se ejecuta inmediatamente después de ingresarlo. Además, debido a que la sesión interactiva imprime automáticamente los resultados de las expresiones que ingresa, generalmente no necesita decir "imprimir" explícitamente en este indicador:

>>> lumberjack = 'okay'

>>> lumberjack

'okay'

>>> 2 \*\* 8

256

>>> # Use Ctrl-D or Ctrl-Z to exit

%

Here, the last two lines typed are expressions (lumberjack and 2 \*\* 8), and their results are displayed automatically. To exit an interactive session like this one and return to your system shell prompt, type Ctrl-D on Unix-like machines; on MS-DOS and Windows systems, type Ctrl-Z to exit. In the IDLE GUI discussed later, either type Ctrl-D or simply close the window. Now, we didn’t do much in this session’s code—just typed some Python print and assignment statements, and a few expressions, which we’ll study in detail later. The main thing to notice is that the interpreter executes the code entered on each line immediately, when the Enter key is pressed.

Aquí, las dos últimas líneas escritas son expresiones (leñador y 2 \*\* 8), y sus resultados se muestran automáticamente. Para salir de una sesión interactiva como esta y volver al indicador de shell de su sistema, escriba Ctrl-D en máquinas similares a Unix; en sistemas MS-DOS y Windows, escriba Ctrl-Z para salir. En la GUI IDLE que se analiza más adelante, escriba Ctrl-D o simplemente cierre la ventana. Ahora, no hicimos mucho en el código de esta sesión, solo escribimos algunas declaraciones de asignación e impresión de Python, y algunas expresiones, que estudiaremos en detalle más adelante. Lo principal a notar es que el intérprete ejecuta el código ingresado en cada línea inmediatamente, cuando se presiona la tecla Enter.

For instance, when we typed the first print statement at the >>> prompt, the output (a Python string) was echoed back right away. There was no need to create a source-code file, and no need to run the code through a compiler and linker first, as you’d normally do when using a language such as C or C++. As you’ll see in later chapters, you can also run multiline statements at the interactive prompt; such a statement runs immediately after you’ve entered all of its lines. Besides typing python in a shell window, you can also begin similar interactive sessions by starting IDLE’s main window or, on Windows, by selecting the “Python (command-line)” menu option from the Start button menu for Python, as shown in Figure 2-1. Both spawn a >>> prompt with equivalent functionality—code is run as it is typed.

Por ejemplo, cuando escribimos la primera declaración de impresión en el indicador >>>, la salida (una cadena de Python) se repitió de inmediato. No hubo necesidad de crear un archivo de código fuente ni de ejecutar primero el código a través de un compilador y un enlazador, como lo haría normalmente cuando usa un lenguaje como C o C++. Como verá en capítulos posteriores, también puede ejecutar sentencias de varias líneas en el indicador interactivo; dicha declaración se ejecuta inmediatamente después de haber ingresado todas sus líneas. Además de escribir python en una ventana de shell, también puede iniciar sesiones interactivas similares iniciando la ventana principal de IDLE o, en Windows, seleccionando la opción de menú "Python (línea de comandos)" del menú del botón Inicio para Python, como se muestra en la Figura 2-1. Ambos generan un aviso >>> con funcionalidad equivalente: el código se ejecuta a medida que se escribe.

**Testing Code at the Interactive Prompt**

Because code is executed immediately, the interactive prompt turns out to be a great place to experiment with the language. It will be used often in this book to demonstrate smaller examples. In fact, this is the first rule of thumb to remember: if you’re ever in doubt about how a piece of Python code works, fire up the interactive command line and try it out to see what happens. Chances are good that you won’t break anything. (You need to know more about system interfaces before you will become dangerous.) Although you won’t do the bulk of your coding in interactive sessions (because the code you type there is not saved), the interactive interpreter is also a great place to test code you’ve put in files. You can import your module files interactively and run tests on the tools they define by typing calls at the interactive prompt. More generally, the interactive prompt is a place to test program components, regardless of their source— you can type calls to linked-in C functions, exercise Java classes under Jython, and more. Partly because of this interactive nature, Python supports an experimental and exploratory programming style you’ll find convenient when getting started.

Debido a que el código se ejecuta de inmediato, el indicador interactivo resulta ser un excelente lugar para experimentar con el lenguaje. Se usará a menudo en este libro para demostrar ejemplos más pequeños. De hecho, esta es la primera regla general que debe recordar: si alguna vez tiene dudas sobre cómo funciona una parte del código de Python, inicie la línea de comando interactiva y pruébela para ver qué sucede. Hay muchas posibilidades de que no rompas nada. (Necesita saber más sobre las interfaces del sistema antes de volverse peligroso). Aunque no hará la mayor parte de su codificación en sesiones interactivas (porque el código que escribe allí no se guarda), el intérprete interactivo también es un gran lugar para probar el código que ha puesto en los archivos. Puede importar sus archivos de módulo de forma interactiva y ejecutar pruebas en las herramientas que definen escribiendo llamadas en el indicador interactivo. En términos más generales, el indicador interactivo es un lugar para probar los componentes del programa, independientemente de su origen: puede escribir llamadas a funciones C vinculadas, ejercitar clases Java en Jython y más. En parte debido a esta naturaleza interactiva, Python admite un estilo de programación experimental y exploratorio que le resultará conveniente al comenzar.

**Using the Interactive Prompt**

Although the interactive prompt is simple to use, there are a few tips that beginners will need to keep in mind:

Aunque el indicador interactivo es fácil de usar, hay algunos consejos que los principiantes deberán tener en cuenta:

• Type Python commands only. First of all, remember that you can only type Python code at the Python prompt, not system commands. There are ways to run system commands from within Python code (e.g., with os.system), but they are not as direct as simply typing the commands themselves.

Escriba solo comandos de Python. En primer lugar, recuerde que solo puede escribir código de Python en el indicador de Python, no en los comandos del sistema. Hay formas de ejecutar los comandos del sistema desde el código de Python (por ejemplo, con os.system), pero no son tan directas como simplemente escribir los comandos.

• print statements are required only in files. Because the interactive interpreter automatically prints the results of expressions, you do not need to type complete print statements interactively. This is a nice feature, but it tends to confuse users when they move on to writing code in files: within a code file, you must use print statements to see your output, because expression results are not automatically echoed. Remember, you must say print in files, but not interactively.

Las declaraciones de impresión solo se requieren en los archivos. Debido a que el intérprete interactivo imprime automáticamente los resultados de las expresiones, no necesita escribir instrucciones de impresión completas de forma interactiva. Esta es una buena característica, pero tiende a confundir a los usuarios cuando pasan a escribir código en archivos: dentro de un archivo de código, debe usar declaraciones de impresión para ver su salida, porque los resultados de la expresión no se repiten automáticamente. Recuerde, debe decir imprimir en archivos, pero no de forma interactiva.

• Don’t indent at the interactive prompt (yet). When typing Python programs, either interactively or into a text file, be sure to start all your unnested statements in column 1 (that is, all the way to the left). If you don’t, Python may print a “SyntaxError” message. Until Chapter 10, all statements you write will be unnested, so this includes everything for now. This seems to be a recurring confusion in introductory Python classes. A leading space generates an error message.

No sangrar en el indicador interactivo (todavía). Al escribir programas de Python, ya sea de forma interactiva o en un archivo de texto, asegúrese de comenzar todas sus declaraciones no anidadas en la columna 1 (es decir, completamente a la izquierda). Si no lo hace, Python puede imprimir un mensaje de "SyntaxError". Hasta el Capítulo 10, todas las declaraciones que escriba no estarán anidadas, por lo que esto incluye todo por ahora. Esto parece ser una confusión recurrente en las clases introductorias de Python. Un espacio inicial genera un mensaje de error.

• Watch out for prompt changes and compound statements. We won’t meet compound (multiline) statements until Chapter 10, but, as a preview, you should know that when typing lines 2 and beyond of a compound statement interactively, the prompt may change. In the simple shell window interface, the interactive prompt changes to ... instead of >>> for lines 2 and beyond; in the IDLE interface, lines after the first are automatically indented. In either case, inserting a blank line (done by hitting the Enter key at the start of a line) is needed to tell interactive Python that you’re done typing the multiline statement; by contrast, blank lines

are ignored in files.

Tenga cuidado con los cambios rápidos y las declaraciones compuestas. No conoceremos declaraciones compuestas (multilínea) hasta el Capítulo 10, pero, como vista previa, debe saber que al escribir las líneas 2 y más allá de una declaración compuesta de forma interactiva, el aviso puede cambiar. En la interfaz de ventana de shell simple, el indicador interactivo cambia a ... en lugar de >>> para las líneas 2 y siguientes; en la interfaz IDLE, las líneas posteriores a la primera se sangran automáticamente. En cualquier caso, se necesita insertar una línea en blanco (al presionar la tecla Intro al comienzo de una línea) para decirle a Python interactivo que terminó de escribir la declaración de varias líneas; por el contrario, las líneas en blanco se ignoran en los archivos.

You’ll see why this matters in Chapter 10. For now, if you happen to come across a ... prompt or a blank line when entering your code, it probably means that you’ve somehow confused interactive Python into thinking you’re typing a multiline statement. Try hitting the Enter key or a Ctrl-C combination to get back to the main prompt. The >>> and ... prompts can also be changed (they are available in the built-in module sys), but I’ll assume they have not been in

the book’s example listings.

Verá por qué esto es importante en el Capítulo 10. Por ahora, si se encuentra con un mensaje... o una línea en blanco al ingresar su código, probablemente significa que de alguna manera ha confundido a Python interactivo para que piense que está escribir una declaración de varias líneas. Intente presionar la tecla Intro o una combinación de Ctrl-C para volver al indicador principal. Los avisos >>> y ... también se pueden cambiar (están disponibles en el sistema del módulo incorporado), pero supondré que no han estado en las listas de ejemplo del libro.

**System Command Lines and Files**

Although the interactive prompt is great for experimenting and testing, it has one big disadvantage: programs you type there go away as soon as the Python interpreter executes them. The code you type interactively is never stored in a file, so you can’t run it again without retyping it from scratch. Cut-and-paste and command recall can help some here, but not much, especially when you start writing larger programs. To cut and paste code from an interactive session, you have to edit out Python prompts, program outputs, and so on. To save programs permanently, you need to write your code in files, which are usually known as modules. Modules are simply text files containing Python statements. Once coded, you can ask the Python interpreter to execute the statements in such a file any number of times, and in a variety of ways—by system command lines, by file icon clicks, by options in the IDLE user interface, and more. Regardless of how it is run, Python executes all the code in a module file from top to bottom each time you run the file.

Aunque el indicador interactivo es excelente para experimentar y probar, tiene una gran desventaja: los programas que escribe allí desaparecen tan pronto como el intérprete de Python los ejecuta. El código que escribe de forma interactiva nunca se almacena en un archivo, por lo que no puede volver a ejecutarlo sin volver a escribirlo desde cero. Cortar y pegar y recordar comandos puede ayudar un poco aquí, pero no mucho, especialmente cuando comienza a escribir programas más grandes. Para cortar y pegar código de una sesión interactiva, debe editar las indicaciones de Python, las salidas del programa, etc. Para guardar programas de forma permanente, debe escribir su código en archivos, que generalmente se conocen como módulos. Los módulos son simplemente archivos de texto que contienen declaraciones de Python. Una vez codificado, puede pedirle al intérprete de Python que ejecute las declaraciones en dicho archivo cualquier cantidad de veces y de varias maneras: mediante líneas de comando del sistema, haciendo clic en el icono del archivo, mediante opciones en la interfaz de usuario IDLE y más. Independientemente de cómo se ejecute, Python ejecuta todo el código en un archivo de módulo de arriba a abajo cada vez que ejecuta el archivo.

Terminology in this domain can vary somewhat. For instance, module files are often referred to as programs in Python—that is, a program is considered to be a series of precoded statements stored in a file for repeated execution. Module files that are run directly are also sometimes called scripts—an informal term meaning a top-level program file. Some reserve the term “module” for a file imported from another file. (More on the meaning of “top-level” and imports in a few moments.) Whatever you call them, the next few sections explore ways to run code typed into module files. In this section, you’ll learn how to run files in the most basic way: by listing their names in a python command line entered at a system prompt. As a first example, start your favorite text editor (e.g., vi, Notepad, or the IDLE editor), and type two Python statements into a text file named spam.py:

La terminología en este dominio puede variar algo. Por ejemplo, los archivos de módulos a menudo se denominan programas en Python, es decir, se considera que un programa es una serie de declaraciones precodificadas almacenadas en un archivo para su ejecución repetida. Los archivos de módulo que se ejecutan directamente también se denominan a veces secuencias de comandos, un término informal que significa un archivo de programa de nivel superior. Algunos reservan el término "módulo" para un archivo importado de otro archivo. (Más sobre el significado de "nivel superior" e importaciones en unos momentos). Como sea que los llame, las siguientes secciones exploran formas de ejecutar código escrito en archivos de módulo. En esta sección, aprenderá cómo ejecutar archivos de la manera más básica: enumerando sus nombres en una línea de comando de Python ingresada en un indicador del sistema. Como primer ejemplo, inicie su editor de texto favorito (por ejemplo, vi, el Bloc de notas o el editor IDLE) y escriba dos declaraciones de Python en un archivo de texto llamado spam.py:

print 2 \*\* 8 # Raise to a power

print 'the bright side ' + 'of life' # + means concatenation

This file contains two Python print statements, and some Python comments to the right. (Text after a # is simply ignored as a human-readable comment, and is not considered part of the statement’s syntax.) Again, ignore the syntax of the code in this file for now. The point to notice is that you’ve typed the code into a file, rather than at the interactive prompt. In the process, you’ve coded a fully functional Python script. Once you’ve saved this text file, you can ask Python to run it by listing its full filename as the first argument to a python command, typed at the system shell prompt:

Este archivo contiene dos declaraciones de impresión de Python y algunos comentarios de Python a la derecha. (El texto después de un # simplemente se ignora como un comentario legible por humanos y no se considera parte de la sintaxis de la declaración). Nuevamente, ignore la sintaxis del código en este archivo por ahora. El punto a tener en cuenta es que ha escrito el código en un archivo, en lugar de en el indicador interactivo. En el proceso, ha codificado un script de Python completamente funcional. Una vez que haya guardado este archivo de texto, puede pedirle a Python que lo ejecute enumerando su nombre de archivo completo como el primer argumento de un comando de python, escrito en el indicador de shell del sistema:

% python spam.py

256

the bright side of life

Here again, you will type such a system shell command in whatever your system provides for command-line entry—a Windows Command Prompt window, an xterm window, or similar. Remember to replace “python” with a full directory path if your PATH setting is not configured. The output of this little script shows up after the command is typed—it’s the result of the two print statements in the text file. Notice that the module file is called spam.py. As for all top-level files, it could also be called simply spam, but files of code you want to import into a client have to end with a .py suffix. We’ll study imports later in this chapter.

Aquí nuevamente, escribirá un comando de shell del sistema de este tipo en lo que sea que su sistema proporcione para la entrada de la línea de comandos: una ventana del símbolo del sistema de Windows, una ventana xterm o similar. Recuerde reemplazar "python" con una ruta de directorio completa si su configuración de RUTA no está configurada. El resultado de este pequeño script aparece después de escribir el comando: es el resultado de las dos declaraciones de impresión en el archivo de texto. Tenga en cuenta que el archivo del módulo se llama spam.py. En cuanto a todos los archivos de nivel superior, también podría llamarse simplemente spam, pero los archivos de código que desea importar a un cliente deben terminar con el sufijo .py. Estudiaremos las importaciones más adelante en este capítulo.

Because you may want to import a file in the future, it’s a good idea to use .py suffixes for most Python files that you code. Also, some text editors detect Python files by their .py suffix; if the suffix is not present, you may not get features like syntax colorization and automatic indentation. Because this scheme uses shell command lines to start Python programs, all the usual shell syntax applies. For instance, you can route the output of a Python script to a file to save it for later use or inspection by using special shell syntax:

Debido a que es posible que desee importar un archivo en el futuro, es una buena idea usar sufijos .py para la mayoría de los archivos de Python que codifique. Además, algunos editores de texto detectan los archivos de Python por su sufijo .py; si el sufijo no está presente, es posible que no obtenga funciones como la colorización de sintaxis y la sangría automática. Debido a que este esquema usa líneas de comando de shell para iniciar programas de Python, se aplica toda la sintaxis de shell habitual. Por ejemplo, puede enrutar la salida de un script de Python a un archivo para guardarlo para su uso o inspección posterior mediante el uso de una sintaxis de shell especial:

% python spam.py > saveit.txt

In this case, the two output lines shown in the prior run are stored in the file saveit.txt instead of being printed. This is generally known as stream redirection; it works for input and output text, and it works on Windows and Unix-like systems. It also has little to do with Python (Python simply supports it), so we will skip further details on redirection here. If you are working on a Windows platform, this example works the same, but the system prompt is normally different:

En este caso, las dos líneas de salida que se muestran en la ejecución anterior se almacenan en el archivo saveit.txt en lugar de imprimirse. Esto generalmente se conoce como redirección de transmisión; funciona para texto de entrada y salida, y funciona en sistemas Windows y similares a Unix. También tiene poco que ver con Python (Python simplemente lo admite), por lo que omitiremos más detalles sobre la redirección aquí. Si está trabajando en una plataforma Windows, este ejemplo funciona igual, pero el indicador del sistema normalmente es diferente:

C:\Python25> python spam.py

256

the bright side of life

As usual, be sure to type the full path to Python if you haven’t set your PATH environment variable, and haven’t run a change-directory command:

Como de costumbre, asegúrese de escribir la ruta completa a Python si no ha configurado su variable de entorno PATH y no ha ejecutado un comando de cambio de directorio:

D:\temp> C:\python25\python spam.py

256

the bright side of life

On newer versions of Windows, you can also type just the name of your script, regardless of the directory in which you’re working. Because newer Windows systems use the Windows Registry to find a program with which to run a file, you don’t need to list it on the command line explicitly. The prior command, for example, could be simplified to this on a recent version of Windows:

En las versiones más recientes de Windows, también puede escribir solo el nombre de su secuencia de comandos, independientemente del directorio en el que esté trabajando. Debido a que los sistemas Windows más nuevos utilizan el Registro de Windows para encontrar un programa con el que ejecutar un archivo, no es necesario que lo incluya en la línea de comandos de forma explícita. El comando anterior, por ejemplo, podría simplificarse a esto en una versión reciente de Windows:

D:\temp> spam.py

Finally, remember to give the full path to your script file if it lives in a different directory from the one in which you are working. For example, the following system command line, run from D:\other, assumes Python is in your system path but runs a file located elsewhere:

Finalmente, recuerde proporcionar la ruta completa a su archivo de script si se encuentra en un directorio diferente al que está trabajando. Por ejemplo, la siguiente línea de comando del sistema, ejecutada desde D:\other, asume que Python está en la ruta de su sistema pero ejecuta un archivo ubicado en otro lugar:

D:\other> python c:\code\myscript.py

**Using Command Lines and Files**

Running program files from system command lines is also a fairly straightforward launch option, especially if you are familiar with command lines in general from prior work. For newcomers, though, here are a few pointers about common beginner traps:

La ejecución de archivos de programa desde las líneas de comandos del sistema también es una opción de inicio bastante sencilla, especialmente si está familiarizado con las líneas de comandos en general debido a trabajos anteriores. Sin embargo, para los recién llegados, aquí hay algunos consejos sobre las trampas comunes para principiantes:

• **Beware of automatic extensions on Windows**.

If you use the Notepad program to code program files on Windows, be careful to pick the type All Files when it comes time to save your file, and give the file a .py suffix explicitly. Otherwise, Notepad will save your file with a .txt extension (e.g., as spam.py.txt), making it difficult to run in some launching schemes. Worse, Windows hides file extensions by default, so unless you have changed your view options, you may not even notice that you’ve coded a text file and not a Python file. The file’s icon may give this away—if it doesn’t have a snake on it, you may have trouble. Uncolored code in IDLE and files that open to edit instead of run when clicked are other symptoms of this problem. Microsoft Word similarly adds a .doc extension by default; much worse, it adds formatting characters that are not legal Python syntax. As a rule of thumb, always pick All Files when saving under Windows, or use more programmer-friendly text editors such as IDLE. IDLE does not even add a .py suffix automatically—a feature programmers tend to like, and users do not.

Si usa el programa Bloc de notas para codificar archivos de programa en Windows, tenga cuidado de elegir el tipo Todos los archivos cuando llegue el momento de guardar su archivo y asigne al archivo un sufijo .py explícitamente. De lo contrario, el Bloc de notas guardará su archivo con una extensión .txt (por ejemplo, como spam.py.txt), lo que dificultará su ejecución en algunos esquemas de inicio. Peor aún, Windows oculta las extensiones de archivo de forma predeterminada, por lo que, a menos que haya cambiado sus opciones de visualización, es posible que ni siquiera note que ha codificado un archivo de texto y no un archivo de Python. El ícono del archivo puede revelar esto: si no tiene una serpiente, es posible que tenga problemas. El código sin colorear en IDLE y los archivos que se abren para editar en lugar de ejecutarse cuando se hace clic son otros síntomas de este problema. Microsoft Word agrega de manera similar una extensión .doc de forma predeterminada; mucho peor, agrega caracteres de formato que no son sintaxis legal de Python. Como regla general, elija siempre Todos los archivos cuando guarde en Windows, o use editores de texto más fáciles de usar como IDLE. IDLE ni siquiera agrega un sufijo .py automáticamente, una característica que suele gustar a los programadores y no a los usuarios.

• Use file extensions at system prompts, but not for imports. Don’t forget to type the full name of your file in system command lines—that is, use python spam.py rather than python spam. Python import statements, which we’ll meet later in this chapter, omit both the .py file suffix, and the directory path (e.g., import spam). This may seem simple, but it’s a common mistake.

Use extensiones de archivo en las indicaciones del sistema, pero no para las importaciones. No olvide escribir el nombre completo de su archivo en las líneas de comando del sistema, es decir, use python spam.py en lugar de python spam. Las instrucciones de importación de Python, que veremos más adelante en este capítulo, omiten tanto el sufijo del archivo .py como la ruta del directorio (por ejemplo, import spam). Esto puede parecer simple, pero es un error común.

At the system prompt, you are in a system shell, not Python, so Python’s module file search rules do not apply. Because of that, you must include the .py extension, and you can optionally include a full directory path leading to the file you wish to run. For instance, to run a file that resides in a different directory from the one in which you are working, you would typically list its full path (i.e., C:\python25>python d:\tests\spam.py). Within Python code, however, you just say import spam, and rely on the Python module search path to locate your file, as described later.

En el indicador del sistema, se encuentra en un shell del sistema, no en Python, por lo que no se aplican las reglas de búsqueda de archivos del módulo de Python. Por eso, debe incluir la extensión .py y, opcionalmente, puede incluir una ruta de directorio completa que conduzca al archivo que desea ejecutar. Por ejemplo, para ejecutar un archivo que se encuentra en un directorio diferente al que está trabajando, normalmente enumeraría su ruta completa (es decir, C:\python25>python d:\tests\spam.py). Sin embargo, dentro del código de Python, solo dice importar spam y confía en la ruta de búsqueda del módulo de Python para ubicar su archivo, como se describe más adelante.

• Use print statements in files. Yes, we’ve already been over this, but it is such a common mistake that it’s worth repeating here. Unlike in interactive coding, you generally must use print statements to see output from program files

Utilice sentencias de impresión en archivos. Sí, ya hemos hablado de esto, pero es un error tan común que vale la pena repetirlo aquí. A diferencia de la codificación interactiva, generalmente debe usar declaraciones de impresión para ver la salida de los archivos de programa

**Unix Executable Scripts (#!)**

If you are going to use Python on a Unix, Linux, or Unix-like system, you can also turn files of Python code into executable programs, much as you would for programs coded in a shell language such as csh or ksh. Such files are usually called executable scripts. In simple terms, Unix-style executable scripts are just normal text files containing Python statements, but with two special properties:

Si va a usar Python en un sistema Unix, Linux o similar a Unix, también puede convertir archivos de código de Python en programas ejecutables, como lo haría con programas codificados en un lenguaje de shell como csh o ksh. Dichos archivos generalmente se denominan scripts ejecutables. En términos simples, los scripts ejecutables de estilo Unix son solo archivos de texto normales que contienen declaraciones de Python, pero con dos propiedades especiales:

• Their first line is special. Scripts usually start with a line that begins with the characters #! (often called “hash bang”), followed by the path to the Python interpreter on your machine.

Su primera línea es especial. Los guiones generalmente comienzan con una línea que comienza con los caracteres #! (a menudo llamado "hash bang"), seguido de la ruta al intérprete de Python en su máquina.

• They usually have executable privileges. Script files are usually marked as executable to tell the operating system that they may be run as top-level programs. On Unix systems, a command such as chmod +x file.py usually does the trick.

Suelen tener privilegios de ejecución. Los archivos de script generalmente se marcan como ejecutables para indicarle al sistema operativo que pueden ejecutarse como programas de nivel superior. En los sistemas Unix, un comando como chmod +x file.py generalmente funciona.

Let’s look at an example for Unix-like systems. Use your text editor again to create a file of Python code called brian:

Veamos un ejemplo para sistemas similares a Unix. Use su editor de texto nuevamente para crear un archivo de código Python llamado brian:

#!/usr/local/bin/python

print 'The Bright Side of Life...' # Another comment here

The special line at the top of the file tells the system where the Python interpreter lives. Technically, the first line is a Python comment. As mentioned earlier, all comments in Python programs start with a # and span to the end of the line; they are a place to insert extra information for human readers of your code. But when a comment such as the first line in this file appears, it’s special because the operating system uses it to find an interpreter for running the program code in the rest of the file. Also, note that this file is called simply brian, without the .py suffix used for the module file earlier. Adding a .py to the name wouldn’t hurt (and might help you remember that this is a Python program file), but because you don’t plan on letting other modules import the code in this file, the name of the file is irrelevant. If you give the file executable privileges with a chmod +x brian shell command, you can run it from the operating system shell as though it were a binary program:

La línea especial en la parte superior del archivo le dice al sistema dónde vive el intérprete de Python. Técnicamente, la primera línea es un comentario de Python. Como se mencionó anteriormente, todos los comentarios en los programas de Python comienzan con un # y se extienden hasta el final de la línea; son un lugar para insertar información adicional para lectores humanos de su código. Pero cuando aparece un comentario como la primera línea de este archivo, es especial porque el sistema operativo lo usa para encontrar un intérprete para ejecutar el código del programa en el resto del archivo. Además, tenga en cuenta que este archivo se llama simplemente brian, sin el sufijo .py utilizado anteriormente para el archivo del módulo. Agregar un .py al nombre no estaría de más (y podría ayudarlo a recordar que se trata de un archivo de programa de Python), pero debido a que no planea permitir que otros módulos importen el código en este archivo, el nombre del archivo es irrelevante. Si otorga al archivo privilegios ejecutables con un comando chmod +x brian shell, puede ejecutarlo desde el shell del sistema operativo como si fuera un programa binario:

% brian

The Bright Side of Life...

A note for Windows users: the method described here is a Unix trick, and it may not work on your platform. Not to worry; just use the basic command-line technique explored earlier. List the file’s name on an explicit python command line:\*

Una nota para los usuarios de Windows: el método descrito aquí es un truco de Unix y es posible que no funcione en su plataforma. No es para preocuparse; solo use la técnica básica de línea de comandos explorada anteriormente. Enumere el nombre del archivo en una línea de comando explícita de Python:\*

C:\book\tests> python brian

The Bright Side of Life...

\* As we discussed when exploring command lines, modern Windows versions also let you type just the name of a .py file at the system command line—they use the Registry to determine that the file should be opened with Python (e.g., typing brian.py is equivalent to typing python brian.py). This command-line mode is similar in spirit to the Unix #!. Note that some programs may actually interpret and use a first #! line on Windows much like on Unix, but the DOS system shell on Windows ignores it completely.

Como discutimos al explorar las líneas de comando, las versiones modernas de Windows también le permiten escribir solo el nombre de un archivo .py en la línea de comando del sistema; usan el Registro para determinar que el archivo debe abrirse con Python (por ejemplo, escribir brian.py es equivalente a escribir python brian.py). Este modo de línea de comandos es similar en espíritu al Unix #!. ¡Tenga en cuenta que algunos programas pueden interpretar y usar un primer #! en Windows como en Unix, pero el shell del sistema DOS en Windows lo ignora por completo.

In this case, you don’t need the special #! comment at the top (although Python just ignores it if it’s present), and the file doesn’t need to be given executable privileges. In fact, if you want to run files portably between Unix and Microsoft Windows, your life will probably be simpler if you always use the basic command-line approach, not Unix-style scripts, to launch programs

En este caso, ¡no necesita el # especial! comentario en la parte superior (aunque Python simplemente lo ignora si está presente), y no es necesario otorgar privilegios de ejecución al archivo. De hecho, si desea ejecutar archivos de forma portátil entre Unix y Microsoft Windows, su vida probablemente será más simple si siempre usa el enfoque básico de línea de comandos, no los scripts de estilo Unix, para iniciar programas.

**The Unix env Lookup Trick**

On some Unix systems, you can avoid hardcoding the path to the Python interpreter by writing the special first-line comment like this:

En algunos sistemas Unix, puede evitar codificar la ruta al intérprete de Python escribiendo un comentario especial en la primera línea como este:

#!/usr/bin/env python

...script goes here...

When coded this way, the env program locates the Python interpreter according to your system search path settings (i.e., in most Unix shells, by looking in all the directories listed in the PATH environment variable). This scheme can be more portable, as you don’t need to hardcode a Python install path in the first line of all your scripts. Provided you have access to env everywhere, your scripts will run no matter where Python lives on your system—you need only change the PATH environment variable settings across platforms, not in the first line in all your scripts. Of course, this assumes that env lives in the same place everywhere (on some machines, it may also be in /sbin, /bin, or elsewhere); if not, all portability bets are off.

Cuando se codifica de esta manera, el programa env ubica el intérprete de Python de acuerdo con la configuración de la ruta de búsqueda de su sistema (es decir, en la mayoría de los shells de Unix, al buscar en todos los directorios enumerados en la variable de entorno PATH). Este esquema puede ser más portátil, ya que no necesita codificar una ruta de instalación de Python en la primera línea de todos sus scripts. Siempre que tenga acceso a env en todas partes, sus secuencias de comandos se ejecutarán sin importar dónde se encuentre Python en su sistema; solo necesita cambiar la configuración de la variable de entorno PATH en todas las plataformas, no en la primera línea de todas sus secuencias de comandos. Por supuesto, esto supone que env vive en el mismo lugar en todas partes (en algunas máquinas, también puede estar en /sbin, /bin o en otro lugar); si no, todas las apuestas de portabilidad están canceladas.

**Clicking File Icons**

On Windows, the Registry makes opening files with icon clicks easy. Python automatically registers itself to be the program that opens Python program files when they are clicked. Because of that, it is possible to launch the Python programs you write by simply clicking (or double-clicking) on their file icons with your mouse. On non-Windows systems, you will probably be able to perform a similar trick, but the icons, file explorer, navigation schemes, and more may differ slightly. On some Unix systems, for instance, you may need to register the .py extension with your file explorer GUI, make your script executable using the #! trick discussed in the prior section, or associate the file MIME type with an application or command by editing files, installing programs, or using other tools. See your file explorer’s documentation for more details if clicks do not work correctly right off the bat.

En Windows, el Registro facilita la apertura de archivos con clics en los iconos. Python se registra automáticamente para ser el programa que abre los archivos de programa de Python cuando se hace clic en ellos. Por eso, es posible iniciar los programas de Python que escribe simplemente haciendo clic (o doble clic) en los iconos de archivo con el mouse. En sistemas que no sean Windows, probablemente podrá realizar un truco similar, pero los íconos, el explorador de archivos, los esquemas de navegación y más pueden diferir ligeramente. En algunos sistemas Unix, por ejemplo, es posible que deba registrar la extensión .py con la GUI de su explorador de archivos, hacer que su secuencia de comandos sea ejecutable usando #! truco discutido en la sección anterior, o asociar el tipo de archivo MIME con una aplicación o comando editando archivos, instalando programas o usando otras herramientas. Consulte la documentación de su explorador de archivos para obtener más detalles si los clics no funcionan correctamente desde el principio.

**Clicking Icons on Windows**

To illustrate, suppose you create the following program file with your text editor and

save it as script4.py:

# A comment

import sys

print sys.platform

print 2 \*\* 100

There’s not much new here—just an import and two prints again (sys.platform is just a string that identifies the kind of computer you’re working on; it lives in a module called sys, which you must import to load). You can run this file from a system command line:

D:\LP3E\Examples> c:\python25\python script4.py

win32

1267650600228229401496703205376

However, icon clicks allow you to run the file without any typing at all. If you find this file’s icon—for instance, by selecting My Computer and working your way down on the D drive—you will get the file explorer picture captured in Figure 3-1 (Windows XP is being used here). In Python 2.5, source files show up with white backgrounds on Windows, and byte code files show up with black backgrounds. You will normally want to click (or otherwise run) the source code file, in order to pick up your most recent changes. To launch the file here, simply click on the icon for script4.py

Sin embargo, los clics en el icono le permiten ejecutar el archivo sin escribir nada. Si encuentra el ícono de este archivo, por ejemplo, seleccionando Mi PC y avanzando hacia abajo en la unidad D, obtendrá la imagen del explorador de archivos capturada en la Figura 3-1 (aquí se usa Windows XP). En Python 2.5, los archivos de origen se muestran con fondos blancos en Windows y los archivos de código de bytes se muestran con fondos negros. Normalmente querrá hacer clic (o ejecutar) el archivo de código fuente para recoger los cambios más recientes. Para iniciar el archivo aquí, simplemente haga clic en el icono de script4.py

**The raw\_input Trick**

Unfortunately, on Windows, the result of clicking on a file icon may not be incredibly satisfying. In fact, as it is, this example script generates a perplexing “flash” when clicked—not the sort of feedback that budding Python programmers usually hope for! This is not a bug, but it has to do with the way the Windows port handles printed output. By default, Python generates a pop-up black DOS console window to serve as a clicked file’s input and output. If a script prints and exits, then, well, it just prints and exits—the console window appears, and text is printed there, but the console window closes and disappears on program exit. Unless you are very fast, or your machine is very slow, you won’t get to see your output at all. Although this is normal behavior, it’s probably not what you had in mind. Luckily, it’s easy to work around this. If you need your script’s output to stick around when you launch it with an icon click, simply put a call to the built-in raw\_input function at the very bottom of the script. For example:

Desafortunadamente, en Windows, el resultado de hacer clic en el ícono de un archivo puede no ser increíblemente satisfactorio. De hecho, tal como está, esta secuencia de comandos de ejemplo genera un "destello" desconcertante cuando se hace clic, ¡no es el tipo de retroalimentación que los programadores de Python en ciernes suelen esperar! Esto no es un error, pero tiene que ver con la forma en que el puerto de Windows maneja la salida impresa. De forma predeterminada, Python genera una ventana de consola de DOS negra emergente que sirve como entrada y salida del archivo en el que se ha hecho clic. Si un script se imprime y se cierra, entonces, bueno, simplemente se imprime y se cierra: aparece la ventana de la consola y el texto se imprime allí, pero la ventana de la consola se cierra y desaparece al salir del programa. A menos que sea muy rápido o que su máquina sea muy lenta, no podrá ver su salida en absoluto. Aunque este es un comportamiento normal, probablemente no sea lo que tenías en mente. Afortunadamente, es fácil solucionar esto. Si necesita que la salida de su secuencia de comandos se mantenga cuando la inicia con un clic en el ícono, simplemente llame a la función raw\_input incorporada en la parte inferior de la secuencia de comandos. Por ejemplo:

# A comment

import sys

print sys.platform

print 2 \*\* 100

raw\_input( ) # ADDED

In general, raw\_input reads the next line of standard input, waiting if there is none yet available. The net effect in this context will be to pause the script, thereby keeping the output window shown in Figure 3-2 open until you press the Enter key.

Now that I’ve shown you this trick, keep in mind that it is usually only required for Windows, and then only if your script prints text and exits, and only if you will launch the script by clicking its file icon. You should add this call to the bottom of your top-level files if and only if all of these three conditions apply. There is no reason to add this call in any other contexts.\*

Before we move ahead, note that the raw\_input call applied here is the input counter-

part of using the print statement for outputs. It is the simplest way to read user

input, and it is more general than this example implies. For instance, raw\_input:

• Optionally accepts a string that will be printed as a prompt (e.g., raw\_input('Press

Enter to exit'))

• Returns to your script the line of text read as a string (e.g., nextinput = raw\_input( ))

• Supports input stream redirections at the system shell level (e.g., python spam.py

< input.txt), just as the print statement does for output

We’ll use raw\_input in more advanced ways later in this text; for instance,

Chapter 10 will apply it in an interactive loop.

Other Icon-Click Limitations

Even with the raw\_input trick, clicking file icons is not without its perils. You also

may not get to see Python error messages. If your script generates an error, the error

message text is written to the pop-up console window—which then immediately dis-

appears. Worse, adding a raw\_input call to your file will not help this time because

your script will likely abort long before it reaches this call. In other words, you won’t

be able to tell what went wrong.

Because of these limitations, it is probably best to view icon clicks as a way to launch

programs after they have been debugged. Especially when starting out, use other

techniques—such as system command lines and IDLE (discussed later in this chap-

ter)—so that you can see generated error messages and view your normal output

without resorting to coding tricks. When we discuss exceptions later in this book,

you’ll also learn that it is possible to intercept and recover from errors so that they do

not terminate your programs. Watch for the discussion of the try statement later in

this book for an alternative way to keep the console window from closing on errors.

Module Imports and Reloads

So far, I’ve been talking about “importing modules” without really explaining what

this term means. We’ll study modules and larger program architecture in depth in

Part V, but because imports are also a way to launch programs, this section will

introduce enough module basics to get you started.

\* It is also possible to completely suppress the pop-up DOS console window for clicked files on Windows.

Files whose names end in a .pyw extension will display only windows constructed by your script, not the

default DOS console window. .pyw files are simply .py source files that have this special operational behavior

on Windows. They are mostly used for Python-coded user interfaces that build windows of their own, often

in conjunction with various techniques for saving printed output and errors to files

46 | Chapter 3: How You Run Programs

In simple terms, every file of Python source code whose name ends in a .py exten-

sion is a module. Other files can access the items a module defines by importing that

module; import operations essentially load another file, and grant access to that file’s

contents. The contents of a module are made available to the outside world through

its attributes (a term I’ll define in the next section).

This module-based services model turns out to be the core idea behind program

architecture in Python. Larger programs usually take the form of multiple module

files, which import tools from other module files. One of the modules is designated

as the main or top-level file, and is the one launched to start the entire program.

We’ll delve into such architectural issues in more detail later in this book. This chap-

ter is mostly interested in the fact that import operations run the code in a file that is

being loaded as a final step. Because of this, importing a file is yet another way to

launch it.

For instance, if you start an interactive session (in IDLE, from a command line, or

otherwise), you can run the script4.py file you created earlier with a simple import:

D:\LP3E\Examples> c:\python25\python

>>> import script4

win32

1267650600228229401496703205376

This works, but only once per session (really, process), by default. After the first

import, later imports do nothing, even if you change and save the module’s source

file again in another window:

>>> import script4

>>> import script4

This is by design; imports are too expensive an operation to repeat more than once

per program run. As you’ll learn in Chapter 18, imports must find files, compile to

byte code, and run the code.

If you really want to force Python to run the file again in the same session (without stop-

ping and restarting the session), you need to instead call the built-in reload function:

>>> reload(script4)

win32

65536

<module 'script4' from 'script4.py'>

>>>

The reload function loads and runs the current version of your file’s code if you’ve

changed it in another window. This allows you to edit and pick up new code on the

fly within the current Python interactive session. In this session, for example, the sec-

ond print statement in script4.py was changed in another window to print 2 \*\* 16

between the time of the first import and the reload call.

Module Imports and Reloads | 47

The reload function expects the name of an already loaded module object, so you

have to have successfully imported a module once before you reload it. Notice that

reload also expects parentheses around the module object name, whereas import

does not. reload is a function that is called, and import is a statement. That’s why

you must pass the module name to reload as an argument in parentheses, and that’s

why you get back an extra output line when reloading. The last output line is just

print’s representation of the reload call’s return value, a Python module object.

Functions will be discussed further in Chapter 15.

The Grander Module Story: Attributes

Imports and reloads provide a natural program launch option because import opera-

tions execute files as a last step. In the broader scheme of things, though, modules

serve the role of libraries of tools, as you’ll learn in Part V. More generally, a module

is mostly just a package of variable names, known as a namespace. The names within

that package are called attributes—that is, an attribute is a variable name that is

attached to a specific object.

In typical use, importers gain access to all the names assigned at the top level of a

module’s file. These names are usually assigned to tools exported by the module—

functions, classes, variables, and so on—that are intended to be used in other files

and other programs. Externally, a module file’s names can be fetched with two

Python statements, import and from, as well as the reload call.

To illustrate, use a text editor to create a one-line Python module file called myfile.py

with the following contents:

title = "The Meaning of Life"

This may be one of the world’s simplest Python modules (it contains a single assign-

ment statement), but it’s enough to illustrate the basics. When this file is imported,

its code is run to generate the module’s attribute. The assignment statement creates a

module attribute named title.

You can access this module’s title attribute in other components in two different

ways. First, you can load the module as a whole with an import statement, and then

qualify the module name with the attribute name to access it:

% python # Start Python

>>> import myfile # Run file; load module as a whole

>>> print myfile.title # Use its attribute names: '.' to qualify

The Meaning of Life

In general, the dot expression syntax object.attribute lets you fetch any attribute

attached to any object, and this is a common operation in Python code. Here, we’ve

used it to access the string variable title inside the module myfile—in other words,

myfile.title.

48 | Chapter 3: How You Run Programs

Alternatively, you can fetch (really, copy) names out of a module with from statements:

% python # Start Python

>>> from myfile import title # Run file; copy its names

>>> print title # Use name directly: no need to qualify

The Meaning of Life

As you’ll see in more detail later, from is just like an import, with an extra assignment

to names in the importing component. Technically, from copies a module’s attributes,

such that they become simple variables in the recipient—thus, you can simply refer to

the imported string this time as title (a variable) instead of myfile.title (an

attribute reference).\*

Whether you use import or from to invoke an import operation, the statements in the

module file myfile.py are executed, and the importing component (here, the interac-

tive prompt) gains access to names assigned at the top level of the file. There’s only

one such name in this simple example—the variable title, assigned to a string—but

the concept will be more useful when you start defining objects such as functions

and classes in your modules. Such objects become reusable software components

that can be accessed by name from one or more client modules.

In practice, module files usually define more than one name to be used in and out-

side the files. Here’s an example that defines three:

a = 'dead' # Define three attributes

b = 'parrot' # Exported to other files

c = 'sketch'

print a, b, c # Also used in this file

This file, threenames.py, assigns three variables, and so generates three attributes for

the outside world. It also uses its own three variables in a print statement, as we see

when we run this as a top-level file:

% python threenames.py

dead parrot sketch

All of this file’s code runs as usual the first time it is imported elsewhere (by either an

import or from). Clients of this file that use import get a module with attributes, while

clients that use from get copies of the file’s names:

% python

>>> import threenames # Grab the whole module

dead parrot sketch

>>>

>>> threenames.b, threenames.c

('parrot', 'sketch')

>>>

\* Notice that import and from both list the name of the module file as simply myfile without its .py suffix. As

you’ll learn in Part V, when Python looks for the actual file, it knows to include the suffix in its search pro-

cedure. Again, you must remember to include the suffix in system shell command lines, but not in import

statements.

Module Imports and Reloads | 49

>>> from threenames import a, b, c # Copy multiple names

>>> b, c

('parrot', 'sketch')

The results here are printed in parentheses because they are really tuples (a kind of

object covered in the next part of this book).

Once you start coding modules with multiple names like this, the built-in dir func-

tion starts to come in handy. You can use it to fetch a list of the names available

inside a module:

>>> dir(threenames)

['\_ \_builtins\_ \_', '\_ \_doc\_ \_', '\_ \_file\_ \_', '\_ \_name\_ \_', 'a', 'b', 'c']

When the dir function is called with the name of an imported module passed in

parentheses like this, it returns all the attributes inside that module. Some of the

names it returns are names you get “for free”: names with leading and trailing dou-

ble underscores are built-in names that are always predefined by Python, and that

have special meaning to the interpreter. The variables our code defined by assign-

ment—a, b, and c—show up last in the dir result.

Modules and namespaces

Module imports are a way to run files of code, but, as we’ll discuss later in the book,

modules are also the largest program structure in Python programs. In general,

Python programs are composed of multiple module files, linked together by import

statements. Each module file is a self-contained package of variables—that is, a

namespace. One module file cannot see the names defined in another file unless it

explicitly imports that other file, so modules serve to minimize name collisions in

your code—because each file is a self-contained namespace, the names in one file

cannot clash with those in another, even if they are spelled the same way.

In fact, as you’ll see, modules are one of a handful of ways that Python goes to great

lengths to package your variables into compartments to avoid name clashes. We’ll

discuss modules and other namespace constructs (including classes and function

scopes) further later in the book. For now, modules will come in handy as a way to

run your code many times without having to retype it.

import and reload Usage Notes

For some reason, once people find out about running files using import and reload,

many tend to focus on this alone and forget about other launch options that always

run the current version of the code (e.g., icon clicks, IDLE menu options, and sys-

tem command lines). This can quickly lead to confusion—you need to remember

when you’ve imported to know if you can reload, you need to remember to use

parentheses when you call reload (only), and you need to remember to use reload in

the first place to get the current version of your code to run.

50 | Chapter 3: How You Run Programs

Because of these complications (and others we’ll meet later), it’s a good idea to avoid

the temptation to launch by imports and reloads for now. The IDLE Run ➝ Run

Module menu option, for example, provides a simpler and less error-prone way to

run your files. On the other hand, imports and reloads have proven to be a popular

testing technique in Python classes. You may prefer using this approach, but if you

find yourself running into a wall, stop.

There is more to the module story than we’ve exposed here. For instance, the

execfile('module.py') built-in function is another way to launch files from the inter-

active prompt without having to import and later reload. It has a similar effect, but

doesn’t technically import the module—by default, each time you call execfile, it

runs the file anew, as though you had pasted it in at the place where execfile is

called. Because of that, execfile, like the from statement mentioned earlier, has the

potential to silently overwrite variables you may currently be using. The basic import

statement, on the other hand, runs the file only once per process, and makes the file

a separate namespace so that it will not change variables in your scope.

In addition, you may run into trouble if you use modules in unusual ways at this

point in the book. For instance, if you want to import a module file that is stored in a

directory other than the one you’re working in, you’ll have to skip ahead to

Chapter 18 and learn about the module search path. For now, if you must import, try

to keep all your files in the directory you are working in to avoid complications.

In case you can’t wait until Chapter 18, the short story is that Python

searches for imported modules in every directory listed in sys.path—a

Python list of directory name strings in the sys module, which is ini-

tialized from a PYTHONPATH environment variable, plus a set of standard

directories. If you want to import from a directory other than the one

you are working in, that directory must generally be listed in your

PYTHONPATH setting. For more details, see Chapter 18.

The IDLE User Interface

So far, we’ve seen how to run Python code with the interactive prompt, system com-

mand lines, icon clicks, and module imports. If you’re looking for something a bit

more visual, IDLE provides a graphical user interface (GUI) for doing Python devel-

opment, and it’s a standard and free part of the Python system. It is usually referred

to as an integrated development environment (IDE), because it binds together various

development tasks into a single view. \*

In short, IDLE is a GUI that lets you edit, run, browse, and debug Python programs,

all from a single interface. Moreover, because IDLE is a Python program that uses the

Tkinter GUI toolkit, it runs portably on most Python platforms, including Microsoft

\* IDLE is officially a corruption of IDE, but it’s really named in honor of Monty Python member Eric Idle.

The IDLE User Interface | 51

Windows, X Windows (for Linux, Unix, and Unix-like platforms), and the Mac OS

(both Classic and OS X). For many, IDLE represents an easy-to-use alternative to

typing command lines, and a less problem-prone alternative to clicking on icons.

IDLE Basics

Let’s jump right into an example. IDLE is easy to start under Windows—it has an

entry in the Start button menu for Python (see Figure 2-1), and it can also be selected

by right-clicking on a Python program icon. On some Unix-like systems, you may

need to launch IDLE’s top-level script from a command line, or, alternatively, by

clicking on the icon for the idle.pyw or idle.py file located in the idlelib subdirectory

of Python’s Lib directory. (On Windows, IDLE is a Python script that currently lives

in C:\Python25\Lib\idlelib. \*)

Figure 3-3 shows the scene after starting IDLE on Windows. The Python shell win-

dow that opens initially is the main window, which runs an interactive session

(notice the >>> prompt). This works like all interactive sessions—code you type here

is run immediately after you type it—and serves as a testing tool.

IDLE uses familiar menus with keyboard shortcuts for most of its operations. To

make (or edit) a source code file under IDLE, open a text edit window: in the main

window, select the File pull-down menu, and pick New Window to open a text edit

window (or Open...to edit an existing file). A new window will appear. This is an

IDLE text edit window, where the code for the file you are creating or changing is

entered and displayed.

Although it may not show up fully in this book, IDLE uses syntax-directed coloriza-

tion for the code typed in both the main window and all text edit windows—

keywords are one color, literals are another, and so on. This helps give you a better

picture of the components in your code.

To run a file of code that you are editing in IDLE, select the file’s text edit window,

pick that window’s Run pull-down menu, and choose the Run Module option listed

there (or use the equivalent keyboard shortcut, given in the menu). Python will let

you know that you need to save your file first if you’ve changed it since it was opened

or last saved.

When run this way, the output of your script and any error messages it may generate

show up back in the main interactive window (the Python shell window). In

Figure 3-3, for example, the last three lines in the window reflect an execution of a

\* IDLE is a Python program that uses the standard library’s Tkinter GUI toolkit to build the IDLE GUI. This

makes IDLE portable, but it also means that you’ll need to have Tkinter support in your Python to use IDLE.

The Windows version of Python has this by default, but some Linux and Unix users may need to install the

appropriate Tkinter support (a yum tkinter command may suffice on some Linux distributions, but see the

installation hints in Appendix A for details). Mac OS X may have everything you need preinstalled, too; look

for an idle command or script on your machine.

52 | Chapter 3: How You Run Programs

script opened in a separate edit window; the “RESTART” message tells us that the

user-code process was restarted to run the edited script, and serves to separate script

output.

Hint of the day: if you want to repeat prior commands in IDLE’s main

interactive window, you can use the Alt-P key combination to scroll

backward through the command history, and Alt-N to scroll forward

(on some Macs, try Ctrl-P and Ctrl-N instead). Your prior commands

will be recalled and displayed, and may be edited and rerun. You can

also recall commands by positioning the cursor on them, or use cut-

and-paste operations, but these tend to be more work. Outside IDLE,

you may be able to recall commands in an interactive session with the

arrow keys on Windows.

Using IDLE

IDLE is free, easy to use, portable, and automatically available on most platforms. I

generally recommend it to Python newcomers because it sugarcoats some of the

details, and does not assume prior experience with system command lines. But it is

also somewhat limited compared to more advanced commercial IDEs. Here is a list

of issues that IDLE beginners should bear in mind

The IDLE User Interface | 53

• You must add “.py” explicitly when saving your files. I mentioned this when

talking about files in general, but it’s a common IDLE stumbling block, espe-

cially for Windows users. IDLE does not automatically add a .py extension to

filenames when files are saved. Be careful to type the .py extension yourself when

saving a file for the first time. If you don’t, you will be able to run your file from

IDLE (and system command lines), but you will not be able to import your file

either interactively or from other modules.

• Run scripts by selecting Run ➝ Run Module in text edit windows, not by inter-

active imports and reloads. Earlier in this chapter, we saw that it’s possible to

run a file by importing it interactively. However, this scheme can grow complex

because you are required to manually reload files after changes. By contrast,

using the Run ➝ Run Module menu option in IDLE always runs the most cur-

rent version of your file. It also prompts you to save it first, if needed (another

common mistake outside IDLE).

• You may still have to reload nested modules. Technically speaking, IDLE’s Run

➝ Run Module menu option always runs the current version of the top-level file

only; imported files may still need to be interactively reloaded when changed. In

general, though, Run ➝ Run Module eliminates common confusions surround-

ing imports. If you choose to use the import and reload technique instead,

remember to use Alt-P/Alt-N key combinations to recall prior commands.

• You can customize IDLE. To change the text fonts and colors in IDLE, select the

Configure option in the Options menu of any IDLE window. You can also cus-

tomize key combination actions, indentation settings, and more; see IDLE’s

Help pull-down menu for more hints.

• There is currently no clear-screen option in IDLE. This seems to be a frequent

request (perhaps because it’s an option available in similar IDEs), and it might

be added eventually. Today, though, there is no way to clear the interactive win-

dow’s text. If you want the window’s text to go away, you can either press and

hold the Enter key, or type a Python loop to print a series of blank lines.

• Tkinter GUI and threaded programs may not work well with IDLE. Because

IDLE is a Python/Tkinter program, it can hang if you use it to run certain types

of advanced Python/Tkinter programs. This has become less of an issue in more

recent versions of IDLE that run user code in one process, and the IDLE GUI

itself in another, but some programs may still hang the GUI. Your code may not

exhibit such problems, but, as a rule of thumb, it’s always safe if you use IDLE

to edit GUI programs, but launch them using other options, such as icon clicks

or system command lines. When in doubt, if your code fails in IDLE, try it out-

side the GUI.

• If connection errors arise, try starting IDLE in single-process mode. Because

IDLE requires communication between its separate user and GUI processes, it

can sometimes have trouble starting up on certain platforms (notably, it fails to

54 | Chapter 3: How You Run Programs

start occasionally on some Windows machines). If you run into such connection

errors, it’s always possible to start IDLE with a system command line that forces it

to run in single-process mode, and therefore avoids communication issues: its -n

command-line flag forces this mode. On Windows, for example, start a Com-

mand Prompt window, and run the system command line idle.py –n from within

the directory C:\Python25\Lib\idlelib (cd there first if needed).

• Beware of some IDLE usability features. IDLE does much to make life easier for

beginners, but some of its tricks won’t apply outside the IDLE GUI. For

instance, IDLE runs your script in IDLE’s environment, so variables in your code

show up automatically in the IDLE interactive session—you don’t always need

to run import commands to access names at the top level of files you’ve already

run. This can be handy, but it can also be confusing, because outside the IDLE

environment, names must always be imported from files to be used.

Advanced IDLE Tools

Besides the basic edit and run functions, IDLE provides more advanced features,

including a point-and-click program debugger, and an object browser. The IDLE

debugger is enabled via the Debug menu, and the object browser via the File menu.

The browser allows you to navigate through the module search path to files and

objects in files; clicking on a file or object opens the corresponding source in a text

edit window.

IDLE debugging is initiated by selecting the Debug ➝ Debugger menu option in the

main window, and then starting your script by selecting the Run ➝ Run Module

option in the text edit window; once the debugger is enabled, you can set break-

points in your code that stop its execution by right-clicking on lines in the text edit

windows, show variable values, and so on. You can also watch program execution

when debugging—the current line of code is noted as you step through your code.

For simpler debugging operations, you can also right-click with your mouse on the

text of an error message to quickly jump to the line of code where the error

occurred—a trick that makes it simple and fast to repair and run again. In addition,

IDLE’s text editor offers a large collection of programmer-friendly tools, including

automatic indentation, advanced text and file search operations, and more. Because

IDLE uses intuitive GUI interactions, you should experiment with the system live to

get a feel for its other tools.

Other IDEs

Because IDLE is free, portable, and a standard part of Python, it’s a nice first develop-

ment tool to become familiar with if you want to use an IDE at all. Again, I recommend

that you use IDLE for this book’s exercises if you’re just starting out, unless you are

Other IDEs | 55

already familiar with a command-line-based development mode. There are, however,

a handful of alternative IDEs for Python developers, some of which are substantially

more powerful and robust than IDLE. Here are some of the most commonly used

IDEs:

Eclipse and PyDev

Eclipse is an advanced open source IDE GUI. Originally developed as a Java

IDE, Eclipse also supports Python development when you install the PyDev (or

similar) plug-in. Eclipse is a popular and powerful option for Python develop-

ment, and it goes well beyond IDLE’s feature set. Its downsides seem to be that

it is a large system to install, and its PyDev plug-in requires a shareware exten-

sions package for some features (including an integrated interactive console) that

is not strictly open source. Still, when you are ready to graduate from IDLE, the

Eclipse/PyDev combination is worth your attention.

Komodo

A full-featured development environment GUI for Python (and other lan-

guages), Komodo includes standard syntax-coloring, text-editing, debugging, and

other features. In addition, Komodo offers many advanced features that IDLE

does not, including project files, source-control integration, regular-expression

debugging, and a drag-and-drop GUI builder that generates Python/Tkinter code

to implement the GUIs you design interactively. At this writing, Komodo is not

free; it is available at http://www.activestate.com.

PythonWin

PythonWin is a free Windows-only IDE for Python that ships as part of

ActiveState’s ActivePython distribution (and may also be fetched separately from

http://www.python.org resources). It is roughly like IDLE, with a handful of use-

ful Windows-specific extensions added; for example, PythonWin has support for

COM objects. Today, IDLE is probably more advanced than PythonWin (for

instance, IDLE’s dual-process architecture more often prevents it from becom-

ing hung). However, PythonWin still offers tools for Windows developers that

IDLE does not. See http://www.activestate.com for more information.

Others

There are roughly half a dozen other well-known IDEs that I’m aware of (e.g.,

WingIDE, PythonCard), and more will probably appear over time. In fact,

almost every programmer-friendly text editor has some sort of support for

Python development these days, whether it be preinstalled or fetched separately.

Emacs and Vim, for instance, also have substantial Python support. Rather than

trying to document all such options here, see the resources available at http://

www.python.org, or run a Google web search for “Python editors”—this should

lead you to a Wiki page that maintains information about many IDE and text-

editor options for Python programming.

Embedding Calls

At this point, we’ve seen how to run code typed interactively, and how to launch

code saved in files with system command lines, Unix executable scripts, icon clicks,

module imports, and IDEs like IDLE. That covers most of the cases you’ll see in this

book.

But, in some specialized domains, Python code may also be run by an enclosing sys-

tem. In such cases, we say that the Python programs are embedded in (i.e., run by)

another program. The Python code itself may be entered into a text file, stored in a

database, fetched from an HTML page, parsed from an XML document, and so on.

But from an operational perspective, another system—not you—may tell Python to

run the code you’ve created. Such an embedded execution mode is commonly used

to support end user customization—a game program, for instance, might allow for

play modifications by running user-accessible embedded Python code at strategic

points in time.

As an example, it’s possible to create and run strings of Python code from a C pro-

gram by calling functions in the Python runtime API (a set of services exported by

the libraries created when Python is compiled on your machine):

#include <Python.h>

...

Py\_Initialize( );

PyRun\_SimpleString("x = brave + sir + robin");

In this C code snippet, a program coded in the C language embeds the Python inter-

preter by linking in its libraries, and passes it a Python assignment statement string

to run. C programs may also gain access to Python objects and process or execute

them using other Python API tools.

This book isn’t about Python/C integration, but you should be aware that, depend-

ing on how your organization plans to use Python, you may or may not be the one

who actually starts the Python programs you create. \* Regardless, you can still likely

use the interactive and file-based launching techniques described here to test code in

isolation from those enclosing systems that may eventually use it.

Frozen Binary Executables

Frozen binary executables, described in the preceding chapter, are packages that

combine your program’s byte code and the Python interpreter into a single execut-

able program. With these, Python programs can be launched in the same ways that

\* See Programming Python (O’Reilly) for more details on embedding Python in C/C++. The embedding API

can call Python functions directly, load modules, and more. Also, note that the Jython system allows Java

programs to invoke Python code using a Java-based API (a Python interpreter class)

Future Possibilities? | 57

you would launch any other executable program (icon clicks, command lines, etc.).

While this option works well for delivery of products, it is not really intended for use

during program development. You normally freeze just before shipping (after devel-

opment is finished). See the prior chapter for more on this option.

Text Editor Launch Options

As mentioned previously, although not full-blown IDE GUIs, most programmer-

friendly text editors have support for editing, and possibly running, Python

programs. Such support may be built in or fetchable on the Web. For instance, if you

are familiar with the Emacs text editor, you can do all your Python editing and

launching from inside the text editor itself. See the text editor resources page at http://

www.python.org/editors for more details, or search the Web for the phrase “Python

editors.”

Other Launch Options

Depending on your platform, there may be additional ways that you can start Python

programs. For instance, on some Macintosh systems, you may be able to drag

Python program file icons onto the Python interpreter icon to make them execute.

And on Windows, you can always start Python scripts with the Run... option in the

Start menu. Finally, the Python standard library has utilities that allow Python pro-

grams to be started by other Python programs (e.g., execfile, os.popen, os.system);

however, these tools are beyond the scope of the present chapter.

Future Possibilities?

Although this chapter reflects current practice, much of it has been both platform-

and time-specific. Indeed, many of the execution and launch details presented arose

during the shelf life of this book’s editions. As with program execution options, it’s

not impossible that new program launch options may arise over time.

New operating systems, and new versions of existing systems, may also provide exe-

cution techniques beyond those outlined here. In general, because Python keeps pace

with such changes, you should be able to launch Python programs in whatever way

makes sense for the machines you use, both now and in the future—be that by draw-

ing on tablet PCs or PDAs, grabbing icons in a virtual reality, or shouting a script’s

name over your coworkers’ conversations.

Implementation changes may also impact launch schemes somewhat (e.g., a full

compiler could produce normal executables that are launched much like frozen bina-

ries today). If I knew what the future truly held, though, I would probably be talking

to a stockbroker instead of writing these words!

58 | Chapter 3: How You Run Programs

Which Option Should I Use?

With all these options, one question naturally arises: which one is best for me? In

general, you should use the IDLE interface for development if you are just getting

started with Python. It provides a user-friendly GUI environment, and can hide some

of the underlying configuration details. It also comes with a platform-neutral text

editor for coding your scripts, and it’s a standard and free part of the Python system.

If, on the other hand, you are an experienced programmer, you might be more com-

fortable with simply the text editor of your choice in one window, and another

window for launching the programs you edit via system command lines and icon

clicks (indeed, this is how your author develops Python programs, but he has a Unix-

biased past). Because development environments are a very subjective choice, I can’t

offer much more in the way of universal guidelines; in general, whatever environ-

ment you like to use will usually be the best for you to use.

Chapter Summary

In this chapter, we’ve looked at common ways to launch Python programs: by run-

ning code typed interactively, and by running code stored in files with system

command lines, file-icon clicks, module imports, and IDE GUIs such as IDLE. We’ve

covered a lot of pragmatic startup territory here. This chapter’s goal was to equip

you with enough information that you can start working along with the code we’ll

start writing in the next part of the book. There, we will start exploring the Python

language itself, beginning with its core data types.

First, though, take the usual chapter quiz to exercise what you’ve learned here.

Because this is the last chapter in this part of the book, it’s followed with a set of

more complete exercises that test your mastery of this entire part’s topics. For help

with the latter set of problems, or just for a refresher, turn to Appendix B.

Chapter Quiz | 59

B R A I N B U I L D E R

Chapter Quiz

1. How can you start an interactive interpreter session?

2. Where do you type a system command line to launch a script file?

3. Name two pitfalls related to clicking file icons on Windows.

4. Why might you need to reload a module?

5. How do you run a script from within IDLE?

6. Name two pitfalls related to using IDLE.

7. What is a namespace, and how does it relate to module files?

Quiz Answers

1. You can start an interactive session on Windows by clicking your Start button,

picking the All Programs option, clicking the Python entry, and selecting the

“Python (command line)” menu option. You can also achieve the same effect on

Windows and other platforms by typing python as a system command line in

your system’s console window (a Command Prompt window on Windows).

Another alternative is to launch IDLE, as its main Python shell window is an

interactive session. If you have not set your system’s PATH variable to find

Python, you may need to cd to where Python is installed, or type its full direc-

tory path instead of just python (e.g., C:\Python25\python on Windows).

2. You type system command lines in whatever your platform provides as a system

console: a Command Prompt window on Windows; an xterm or terminal win-

dow on Unix, Linux, and Mac OS X; and so on.

3. Scripts that print and then exit cause the output file to disappear immediately,

before you can view the output (which is why the raw\_input trick comes in

handy); error messages generated by your script also appear in an output

window that closes before you can examine its contents (which is why system

command lines and IDEs such as IDLE are better for most development).

4. Python only imports (loads) a module once per process, by default, so if you’ve

changed its source code and want to run the new version without stopping and

restarting Python, you’ll have to reload it. You must import a module at least

once before you can reload it. Running code from a system command line, or via

an icon click, or an IDE such as IDLE generally makes this a nonissue, as those

launch schemes usually run the current version of the source code file each time.

5. Within the text edit window of the file you wish to run, select the window’s Run

➝ Run Module menu option. This runs the window’s source code as a top-level

script file, and displays its output back in the interactive Python shell window.

60 | Chapter 3: How You Run Programs

6. IDLE can still be hung by some types of programs—especially GUI programs

that perform multithreading (an advanced technique beyond this book’s scope).

Also, IDLE has some usability features that can burn you once you leave the

IDLE GUI: a script’s variables are automatically imported to the interactive

scope in IDLE, for instance, but not by Python in general.

7. A namespace is just a package of variables (i.e., names). It takes the form of an

object with attributes in Python. Each module file is automatically a namespace—

that is, a package of variables reflecting the assignments made at the top level of

the file. Namespaces help avoid name collisions in Python programs: because

each module file is a self-contained namespace, files must explicitly import other

files in order to use their names

Part I Exercises | 61

B R A I N B U I L D E R

Part I Exercises

It’s time to start doing a little coding on your own. This first exercise session is fairly

simple, but a few of these questions hint at topics to come in later chapters. Be sure

to check “Part I, Getting Started” in the solutions appendix (Appendix B) for the

answers; the exercises and their solutions sometimes contain supplemental information

not discussed in the main text of the part, so you should take a peek at the solutions

even if you manage to answer all the questions on your own.

1. Interaction. Using a system command line, IDLE, or another method, start the

Python interactive command line (>>> prompt), and type the expression "Hello

World!" (including the quotes). The string should be echoed back to you. The

purpose of this exercise is to get your environment configured to run Python. In

some scenarios, you may need to first run a cd shell command, type the full path

to the Python executable, or add its path to your PATH environment variable. If

desired, you can set PATH in your .cshrc or .kshrc file to make Python perma-

nently available on Unix systems; on Windows, use a setup.bat, autoexec.bat, or

the environment variable GUI. See Appendix A for help with environment vari-

able settings.

2. Programs. With the text editor of your choice, write a simple module file contain-

ing the single statement print 'Hello module world!' and store it as module1.py.

Now, run this file by using any launch option you like: running it in IDLE, click-

ing on its file icon, passing it to the Python interpreter program on the system

shell’s command line (e.g., python module1.py), and so on. In fact, experiment by

running your file with as many of the launch techniques discussed in this chap-

ter as you can. Which technique seems easiest? (There is no right answer to this

one.)

3. Modules. Next, start the Python interactive command line (>>> prompt) and

import the module you wrote in exercise 2. Try moving the file to a different

directory and importing it again from its original directory (i.e., run Python in

the original directory when you import). What happens? (Hint: is there still a

module1.pyc byte code file in the original directory?)

4. Scripts. If your platform supports it, add the #! line to the top of your module1.py

module file, give the file executable privileges, and run it directly as an execut-

able. What does the first line need to contain? #! usually only has meaning on

Unix, Linux, and Unix-like platforms such as Mac OS X; if you’re working on

Windows, instead try running your file by listing just its name in a DOS console

window without the word “python” before it (this works on recent versions of

Windows), or via the Start ➝ Run...dialog box.

62 | Chapter 3: How You Run Programs

5. Errors. Experiment with typing mathematical expressions and assignments at

the Python interactive command line. First type the expression 1 / 0. What

happens? Next, type a variable name to which you haven’t yet assigned a value.

What happens this time?

You may not know it yet, but you’re doing exception processing (a topic we’ll

explore in depth in Part VII). As you’ll learn there, you are technically triggering

what’s known as the default exception handler—logic that prints a standard error

message. If you do not catch an error, the default handler does and prints the

standard error message in response.

For full-blown source code debugging chores, IDLE includes a GUI debugging

interface (introduced in the “Advanced IDLE Tools” section of this chapter), and

a Python standard library module named pdb provides a command-line debug-

ging interface (you can find more on pdb in the standard library manual). When

you’re first starting out, Python’s default error messages will probably be as

much error handling as you need—they give the cause of the error, as well as

showing the lines in your code that were active when the error occurred.

6. Breaks. At the Python command line, type:

L = [1, 2]

L.append(L)

L

What happens? If you’re using a Python newer than Release 1.5, you’ll probably

see a strange output that we’ll describe in the next part of the book. If you’re

using a Python version older than 1.5.1, a Ctrl-C key combination will probably

help on most platforms. Why do you think this occurs? What does Python

report when you type the Ctrl-C key combination?

If you do have a Python older than Release 1.5.1, make sure your

machine can stop a program with a break-key combination of some

sort before running this test, or you may be waiting a long time.

7. Documentation. Spend at least 17 minutes browsing the Python library and

language manuals before moving on to get a feel for the available tools in the

standard library and the structure of the documentation set. It takes at least this

long to become familiar with the locations of major topics in the manual set;

once you’ve done this, it’s easy to find what you need. You can find this manual

via the Python Start button entry on Windows, in the Python Docs option on the

Help pull-down menu in IDLE, or online at http://www.python.org/doc. I’ll also

have a few more words to say about the manuals and other documentation

sources available (including PyDoc and the help function) in Chapter 14. If you

still have time, go explore the Python web site, as well as the Vaults of Parnassus

and the PyPy third-party extension’s web site. Especially check out the Python.org

documentation and search pages; they can be crucial resources.