maintained, and reused. When using Python, there is no need to start every new script from scratch.

Moreover, we'll find that Python not only includes all the interfaces we need in order to write system tools, but it also fosters script *portability*. By employing Python's standard library, most system scripts written in Python are automatically portable to all major platforms. For instance, you can usually run in Linux a Python directory-processing script written in Windows without changing its source code at all—simply copy over the source code. Though writing scripts that achieve such portability utopia requires some extra effort and practice, if used well, Python could be the only system scripting tool you need to use.

The Next Five Chapters

To make this part of the book easier to study, I have broken it down into five chapters:

- In this chapter, I'll introduce the main system-related modules in overview fashion. We'll meet some of the most commonly used system tools here for the first time.
- In Chapter 3, we continue exploring the basic system interfaces by studying their role in core system programming concepts: streams, command-line arguments, environment variables, and so on.
- Chapter 4 focuses on the tools Python provides for processing files, directories, and directory trees.
- In Chapter 5, we'll move on to cover Python's standard tools for parallel processing—processes, threads, queues, pipes, signals, and more.
- Chapter 6 wraps up by presenting a collection of complete system-oriented programs. The examples here are larger and more realistic, and they use the tools introduced in the prior four chapters to perform real, practical tasks. This collection includes both general system scripts, as well as scripts for processing directories of files.

Especially in the examples chapter at the end of this part, we will be concerned as much with system interfaces as with general Python development concepts. We'll see non-object-oriented and object-oriented versions of some examples along the way, for instance, to help illustrate the benefits of thinking in more strategic ways.

"Batteries Included"

This chapter, and those that follow, deal with both the Python language and its *standard library*—a collection of precoded modules written in Python and C that are automatically installed with the Python interpreter. Although Python itself provides an easy-to-use scripting language, much of the real action in Python development involves this vast library of programming tools (a few hundred modules at last count) that ship with the Python package.

In fact, the standard library is so powerful that it is not uncommon to hear Python described as *batteries included*—a phrase generally credited to Frank Stajano meaning that most of what you need for real day-to-day work is already there for importing. Python's standard library, while not part of the core language per se, is a standard part of the Python system and you can expect it to be available wherever your scripts run. Indeed, this is a noteworthy difference between Python and some other scripting languages—because Python comes with so many library tools "out of the box," supplemental sites like Perl's CPAN are not as important.

As we'll see, the standard library forms much of the challenge in Python programming. Once you've mastered the core language, you'll find that you'll spend most of your time applying the built-in functions and modules that come with the system. On the other hand, libraries are where most of the fun happens. In practice, programs become most interesting when they start using services external to the language interpreter: networks, files, GUIs, XML, databases, and so on. All of these are supported in the Python standard library.

Beyond the standard library, there is an additional collection of *third-party packages* for Python that must be fetched and installed separately. As of this writing, you can find most of these third-party extensions via general web searches, and using the links at *http://www.python.org* and at the PyPI website (accessible from *http://www.python.org*). Some third-party extensions are large systems in their own right; NumPy, Django, and VPython, for instance, add vector processing, website construction, and visualization, respectively.

If you have to do something special with Python, chances are good that either its support is part of the standard Python install package or you can find a free and open source module that will help. Most of the tools we'll employ in this text are a standard part of Python, but I'll be careful to point out things that must be installed separately. Of course, Python's extreme code reuse idiom also makes your programs dependent on the code you reuse; in practice, though, and as we'll see repeatedly in this book, powerful libraries coupled with open source access speed development without locking you into an existing set of features or limitations.

System Scripting Overview

To begin our exploration of the systems domain, we will take a quick tour through the standard library `sys` and `os` modules in this chapter, before moving on to larger system programming concepts. As you can tell from the length of their attribute lists, both of these are large modules—the following reflects Python 3.1 running on Windows 7 outside IDLE:

```
C:\...\PP4E\System> python
Python 3.1.1 (r311:74483, Aug 17 2009, 17:02:12) [MSC v.1500 32 bit (...)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> import sys, os
>>> len(dir(sys))          # 65 attributes
65
```

```
>>> len(dir(os))           # 122 on Windows, more on Unix
122
>>> len(dir(os.path))      # a nested module within os
52
```

The content of these two modules may vary per Python version and platform. For example, `os` is much larger under Cygwin after building Python 3.1 from its source code there (Cygwin is a system that provides Unix-like functionality on Windows; it is discussed further in "More on Cygwin Python for Windows" on page 185):

```
$ ./python.exe
Python 3.1.1 (r311:74480, Feb 20 2010, 10:16:52)
[GCC 3.4.4 (cygming special, gdc 0.12, using dmd 0.125)] on cygwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import sys, os
>>> len(dir(sys))
64
>>> len(dir(os))
217
>>> len(dir(os.path))
51
```

As I'm not going to demonstrate every item in every built-in module, the first thing I want to do is show you how to get more details on your own. Officially, this task also serves as an excuse for introducing a few core system scripting concepts; along the way, we'll code a first script to format documentation.

Python System Modules

Most system-level interfaces in Python are shipped in just two modules: `sys` and `os`. That's somewhat oversimplified; other standard modules belong to this domain too. Among them are the following:

`glob`
: For filename expansion

`socket`
: For network connections and Inter-Process Communication (IPC)

`threading`, `_thread`, `queue`
: For running and synchronizing concurrent threads

`time`, `timeit`
: For accessing system time details

`subprocess`, `multiprocessing`
: For launching and controlling parallel processes

`signal`, `select`, `shutil`, `tempfile`, *and others*
: For various other system-related tasks

Third-party extensions such as pySerial (a serial port interface), Pexpect (an Expect work-alike for controlling cross-program dialogs), and even Twisted (a networking

framework) can be arguably lumped into the systems domain as well. In addition, some built-in functions are actually system interfaces as well—the `open` function, for example, interfaces with the file system. But by and large, `sys` and `os` together form the core of Python's built-in system tools arsenal.

In principle at least, `sys` exports components related to the Python *interpreter* itself (e.g., the module search path), and `os` contains variables and functions that map to the operating system on which Python is run. In practice, this distinction may not always seem clear-cut (e.g., the standard input and output streams show up in `sys`, but they are arguably tied to operating system paradigms). The good news is that you'll soon use the tools in these modules so often that their locations will be permanently stamped on your memory.*

The `os` module also attempts to provide a *portable* programming interface to the underlying operating system; its functions may be implemented differently on different platforms, but to Python scripts, they look the same everywhere. And if that's still not enough, the `os` module also exports a nested submodule, `os.path`, which provides a portable interface to file and directory processing tools.

Module Documentation Sources

As you can probably deduce from the preceding paragraphs, learning to write system scripts in Python is mostly a matter of learning about Python's system modules. Luckily, there are a variety of information sources to make this task easier—from module attributes to published references and books.

For instance, if you want to know everything that a built-in module exports, you can read its library manual entry; study its source code (Python is open source software, after all); or fetch its attribute list and documentation string interactively. Let's import `sys` in Python 3.1 and see what it has to offer:

```
C:\...\PP4E\System> python
>>> import sys
>>> dir(sys)
['__displayhook__', '__doc__', '__excepthook__', '__name__', '__package__',
'__stderr__', '__stdin__', '__stdout__', '_clear_type_cache', '_current_frames',
'_getframe', 'api_version', 'argv', 'builtin_module_names', 'byteorder',
'call_tracing', 'callstats', 'copyright', 'displayhook', 'dllhandle',
'dont_write_bytecode', 'exc_info', 'excepthook', 'exec_prefix', 'executable',
'exit', 'flags', 'float_info', 'float_repr_style', 'getcheckinterval',
'getdefaultencoding', 'getfilesystemencoding', 'getprofile', 'getrecursionlimit',
'getrefcount', 'getsizeof', 'gettrace', 'getwindowsversion', 'hexversion',
'int_info', 'intern', 'maxsize', 'maxunicode', 'meta_path', 'modules', 'path',
'path_hooks', 'path_importer_cache', 'platform', 'prefix', 'ps1', 'ps2',
'setcheckinterval', 'setfilesystemencoding', 'setprofile', 'setrecursionlimit',
```

* They may also work their way into your subconscious. Python newcomers sometimes describe a phenomenon in which they "dream in Python" (insert overly simplistic Freudian analysis here...).

```
'settrace', 'stderr', 'stdin', 'stdout', 'subversion', 'version', 'version_info',
'warnoptions', 'winver']
```

The `dir` function simply returns a list containing the string names of all the attributes in any object with attributes; it's a handy memory jogger for modules at the interactive prompt. For example, we know there is something called `sys.version`, because the name `version` came back in the `dir` result. If that's not enough, we can always consult the `__doc__` string of built-in modules:

```
>>> sys.__doc__
"This module provides access to some objects used or maintained by the\ninterpre
ter and to functions that interact strongly with the interpreter.\n\nDynamic obj
ects:\n\nargv -- command line arguments; argv[0] is the script pathname if known
\npath -- module search path; path[0] is the script directory, else ''\nmodules
-- dictionary of loaded modules\n\ndisplayhook -- called to show results in an i
...lots of text deleted here...
```

Paging Documentation Strings

The `__doc__` built-in attribute just shown usually contains a string of documentation, but it may look a bit weird when displayed this way—it's one long string with embedded end-line characters that print as `\n`, not as a nice list of lines. To format these strings for a more humane display, you can simply use a `print` function-call statement:

```
>>> print(sys.__doc__)
This module provides access to some objects used or maintained by the
interpreter and to functions that interact strongly with the interpreter.

Dynamic objects:

argv -- command line arguments; argv[0] is the script pathname if known
path -- module search path; path[0] is the script directory, else ''
modules -- dictionary of loaded modules

...lots of lines deleted here...
```

The `print` built-in function, unlike interactive displays, interprets end-line characters correctly. Unfortunately, `print` doesn't, by itself, do anything about scrolling or paging and so can still be unwieldy on some platforms. Tools such as the built-in `help` function can do better:

```
>>> help(sys)
Help on built-in module sys:

NAME
    sys

FILE
    (built-in)

MODULE DOCS
    http://docs.python.org/library/sys
```

```
    DESCRIPTION
        This module provides access to some objects used or maintained by the
        interpreter and to functions that interact strongly with the interpreter.

        Dynamic objects:

        argv -- command line arguments; argv[0] is the script pathname if known
        path -- module search path; path[0] is the script directory, else ''
        modules -- dictionary of loaded modules
```

...lots of lines deleted here...

The `help` function is one interface provided by the PyDoc system—standard library code that ships with Python and renders documentation (documentation strings, as well as structural details) related to an object in a formatted way. The format is either like a Unix manpage, which we get for `help`, or an HTML page, which is more grandiose. It's a handy way to get basic information when working interactively, and it's a last resort before falling back on manuals and books.

A Custom Paging Script

The `help` function we just met is also fairly fixed in the way it displays information; although it attempts to page the display in some contexts, its page size isn't quite right on some of the machines I use. Moreover, it doesn't page at all in the IDLE GUI, instead relying on manual use if the scrollbar—potentially painful for large displays. When I want more control over the way help text is printed, I usually use a utility script of my own, like the one in Example 2-1.

Example 2-1. PP4E\System\more.py

```
"""
split and interactively page a string or file of text
"""

def more(text, numlines=15):
    lines = text.splitlines()               # like split('\n') but no '' at end
    while lines:
        chunk = lines[:numlines]
        lines = lines[numlines:]
        for line in chunk: print(line)
        if lines and input('More?') not in ['y', 'Y']: break

if __name__ == '__main__':
    import sys                              # when run, not imported
    more(open(sys.argv[1]).read(), 10)      # page contents of file on cmdline
```

The meat of this file is its `more` function, and if you know enough Python to be qualified to read this book, it should be fairly straightforward. It simply splits up a string around end-line characters, and then slices off and displays a few lines at a time (15 by default) to avoid scrolling off the screen. A slice expression, `lines[:15]`, gets the first 15 items in a list, and `lines[15:]` gets the rest; to show a different number of lines each time,

pass a number to the `numlines` argument (e.g., the last line in Example 2-1 passes 10 to the `numlines` argument of the `more` function).

The `splitlines` string object method call that this script employs returns a list of substrings split at line ends (e.g., `["line", "line",...]`). An alternative `splitlines` method does similar work, but retains an empty line at the end of the result if the last line is `\n` terminated:

```
>>> line = 'aaa\nbbb\nccc\n'

>>> line.split('\n')
['aaa', 'bbb', 'ccc', '']

>>> line.splitlines()
['aaa', 'bbb', 'ccc']
```

As we'll see more formally in Chapter 4, the end-of-line character is normally always `\n` (which stands for a byte usually having a binary value of 10) within a Python script, no matter what platform it is run upon. (If you don't already know why this matters, DOS `\r` characters in text are dropped by default when read.)

String Method Basics

Now, Example 2-1 is a simple Python program, but it already brings up three important topics that merit quick detours here: it uses string methods, reads from a file, and is set up to be run or imported. Python string methods are not a system-related tool per se, but they see action in most Python programs. In fact, they are going to show up throughout this chapter as well as those that follow, so here is a quick review of some of the more useful tools in this set. String methods include calls for searching and replacing:

```
>>> mystr = 'xxxSPAMxxx'
>>> mystr.find('SPAM')                          # return first offset
3
>>> mystr = 'xxaaxxaa'
>>> mystr.replace('aa', 'SPAM')                 # global replacement
'xxSPAMxxSPAM'
```

The `find` call returns the offset of the first occurrence of a substring, and `replace` does global search and replacement. Like all string operations, `replace` returns a new string instead of changing its subject in-place (recall that strings are immutable). With these methods, substrings are just strings; in Chapter 19, we'll also meet a module called `re` that allows regular expression *patterns* to show up in searches and replacements.

In more recent Pythons, the `in` membership operator can often be used as an alternative to `find` if all we need is a yes/no answer (it tests for a substring's presence). There are also a handful of methods for removing whitespace on the ends of strings—especially useful for lines of text read from a file:

```
>>> mystr = 'xxxSPAMxxx'
>>> 'SPAM' in mystr                             # substring search/test
```

```
True
>>> 'Ni' in mystr                          # when not found
False
>>> mystr.find('Ni')
-1

>>> mystr = '\t  Ni\n'
>>> mystr.strip()                          # remove whitespace
'Ni'
>>> mystr.rstrip()                        # same, but just on right side
'\t  Ni'
```

String methods also provide functions that are useful for things such as case conversions, and a standard library module named `string` defines some useful preset variables, among other things:

```
>>> mystr = 'SHRUBBERY'
>>> mystr.lower()                         # case converters
'shrubbery'

>>> mystr.isalpha()                       # content tests
True
>>> mystr.isdigit()
False

>>> import string                         # case presets: for 'in', etc.
>>> string.ascii_lowercase
'abcdefghijklmnopqrstuvwxyz'

>>> string.whitespace                     # whitespace characters
' \t\n\r\x0b\x0c'
```

There are also methods for splitting up strings around a substring delimiter and putting them back together with a substring in between. We'll explore these tools later in this book, but as an introduction, here they are at work:

```
>>> mystr = 'aaa,bbb,ccc'
>>> mystr.split(',')                      # split into substrings list
['aaa', 'bbb', 'ccc']

>>> mystr = 'a  b\nc\nd'
>>> mystr.split()                         # default delimiter: whitespace
['a', 'b', 'c', 'd']

>>> delim = 'NI'
>>> delim.join(['aaa', 'bbb', 'ccc'])     # join substrings list
'aaaNIbbbNIccc'

>>> ' '.join(['A', 'dead', 'parrot'])     # add a space between
'A dead parrot'
```

```
>>> chars = list('Lorreta')                # convert to characters list
>>> chars
['L', 'o', 'r', 'r', 'e', 't', 'a']
>>> chars.append('!')
>>> ''.join(chars)                         # to string: empty delimiter
'Lorreta!'
```

These calls turn out to be surprisingly powerful. For example, a line of data columns separated by tabs can be parsed into its columns with a single `split` call; the *more.py* script uses the `splitlines` variant shown earlier to split a string into a list of line strings. In fact, we can emulate the `replace` call we saw earlier in this section with a split/join combination:

```
>>> mystr = 'xxaaxxaa'
>>> 'SPAM'.join(mystr.split('aa'))         # str.replace, the hard way!
'xxSPAMxxSPAM'
```

For future reference, also keep in mind that Python doesn't automatically convert strings to numbers, or vice versa; if you want to use one as you would use the other, you must say so with manual conversions:

```
>>> int("42"), eval("42")                  # string to int conversions
(42, 42)

>>> str(42), repr(42)                      # int to string conversions
('42', '42')

>>> ("%d" % 42), '{:d}'.format(42)         # via formatting expression, method
('42', '42')

>>> "42" + str(1), int("42") + 1           # concatenation, addition
('421', 43)
```

In the last command here, the first expression triggers string concatenation (since both sides are strings), and the second invokes integer addition (because both objects are numbers). Python doesn't assume you meant one or the other and convert automatically; as a rule of thumb, Python tries to avoid magic—and the temptation to guess—whenever possible. String tools will be covered in more detail later in this book (in fact, they get a full chapter in Part V), but be sure to also see the library manual for additional string method tools.

Other String Concepts in Python 3.X: Unicode and bytes

Technically speaking, the Python 3.X string story is a bit richer than I've implied here. What I've shown so far is the `str` object type—a sequence of characters (technically, Unicode "code points" represented as Unicode "code units") which represents both ASCII and wider Unicode text, and handles encoding and decoding both manually on request and automatically on file transfers. Strings are coded in quotes (e.g., `'abc'`), along with various syntax for coding non-ASCII text (e.g., `'\xc4\xe8'`, `'\u00c4\u00e8'`).

Really, though, 3.X has two additional string types that support most `str` string operations: `bytes`—a sequence of short integers for representing 8-bit binary data, and `bytearray`—a mutable variant of bytes. You generally know you are dealing with `bytes` if strings display or are coded with a leading "b" character before the opening quote (e.g., `b'abc'`, `b'\xc4\xe8'`). As we'll see in Chapter 4, files in 3.X follow a similar dichotomy, using `str` in text mode (which also handles Unicode encodings and line-end conversions) and `bytes` in binary mode (which transfers bytes to and from files unchanged). And in Chapter 5, we'll see the same distinction for tools like sockets, which deal in byte strings today.

Unicode text is used in Internationalized applications, and many of Python's binary-oriented tools deal in byte strings today. This includes some file tools we'll meet along the way, such as the `open` call, and the `os.listdir` and `os.walk` tools we'll study in upcoming chapters. As we'll see, even simple directory tools sometimes have to be aware of Unicode in file content and names. Moreover, tools such as object pickling and binary data parsing are byte-oriented today.

Later in the book, we'll also find that Unicode also pops up today in the text displayed in GUIs; the bytes shipped other networks; Internet standard such as email; and even some persistence topics such as DBM files and shelves. Any interface that deals in text necessarily deals in Unicode today, because `str` *is* Unicode, whether ASCII or wider. Once we reach the realm of the applications programming presented in this book, Unicode is no longer an optional topic for most Python 3.X programmers.

In this book, we'll defer further coverage of Unicode until we can see it in the context of application topics and practical programs. For more fundamental details on how 3.X's Unicode text and binary data support impact both string and file usage in some roles, please see *Learning Python*, Fourth Edition; since this is officially a core language topic, it enjoys in-depth coverage and a full 45-page dedicated chapter in that book.

File Operation Basics

Besides processing strings, the *more.py* script also uses files—it opens the external file whose name is listed on the command line using the built-in `open` function, and it reads that file's text into memory all at once with the file object `read` method. Since file objects returned by `open` are part of the core Python language itself, I assume that you have at least a passing familiarity with them at this point in the text. But just in case you've flipped to this chapter early on in your Pythonhood, the following calls load a file's contents into a string, load a fixed-size set of bytes into a string, load a file's contents into a list of line strings, and load the next line in the file into a string, respectively:

```
open('file').read()           # read entire file into string
open('file').read(N)          # read next N bytes into string
open('file').readlines()      # read entire file into line strings list
open('file').readline()       # read next line, through '\n'
```

As we'll see in a moment, these calls can also be applied to shell commands in Python to read their output. File objects also have `write` methods for sending strings to the associated file. File-related topics are covered in depth in Chapter 4, but making an output file and reading it back is easy in Python:

```
>>> file = open('spam.txt', 'w')        # create file spam.txt
>>> file.write(('spam' * 5) + '\n')     # write text: returns #characters written
21
>>> file.close()

>>> file = open('spam.txt')             # or open('spam.txt').read()
>>> text = file.read()                  # read into a string
>>> text
'spamspamspamspamspam\n'
```

Using Programs in Two Ways

Also by way of review, the last few lines in the *more.py* file in Example 2-1 introduce one of the first big concepts in shell tool programming. They instrument the file to be used in either of two ways—as a *script* or as a *library*.

Recall that every Python module has a built-in `__name__` variable that Python sets to the `__main__` string only when the file is run as a program, not when it's imported as a library. Because of that, the `more` function in this file is executed automatically by the last line in the file when this script is run as a top-level program, but not when it is imported elsewhere. This simple trick turns out to be one key to writing reusable script code: by coding program logic as *functions* rather than as top-level code, you can also import and reuse it in other scripts.

The upshot is that we can run *more.py* by itself or import and call its `more` function elsewhere. When running the file as a top-level program, we list on the command line the name of a file to be read and paged: as I'll describe in more depth in the next chapter, words typed in the command that is used to start a program show up in the built-in `sys.argv` list in Python. For example, here is the script file in action, paging itself (be sure to type this command line in your *PP4E\System* directory, or it won't find the input file; more on command lines later):

```
C:\...\PP4E\System> python more.py more.py
"""
split and interactively page a string or file of text
"""

def more(text, numlines=15):
    lines = text.splitlines()               # like split('\n') but no '' at end
    while lines:
        chunk = lines[:numlines]
        lines = lines[numlines:]
        for line in chunk: print(line)
More?y
        if lines and input('More?') not in ['y', 'Y']: break
```

```
if __name__ == '__main__':
    import sys                              # when run, not imported
    more(open(sys.argv[1]).read(), 10)      # page contents of file on cmdline
```

When the *more.py* file is imported, we pass an explicit string to its `more` function, and this is exactly the sort of utility we need for documentation text. Running this utility on the `sys` module's documentation string gives us a bit more information in human-readable form about what's available to scripts:

```
C:\...\PP4E\System> python
>>> from more import more
>>> import sys
>>> more(sys.__doc__)
This module provides access to some objects used or maintained by the
interpreter and to functions that interact strongly with the interpreter.

Dynamic objects:

argv -- command line arguments; argv[0] is the script pathname if known
path -- module search path; path[0] is the script directory, else ''
modules -- dictionary of loaded modules

displayhook -- called to show results in an interactive session
excepthook -- called to handle any uncaught exception other than SystemExit
  To customize printing in an interactive session or to install a custom
  top-level exception handler, assign other functions to replace these.

stdin -- standard input file object; used by input()
More?
```

Pressing "y" or "Y" here makes the function display the next few lines of documentation, and then prompt again, unless you've run past the end of the lines list. Try this on your own machine to see what the rest of the module's documentation string looks like. Also try experimenting by passing a different window size in the second argument—`more(sys.__doc__, 5)` shows just 5 lines at a time.

Python Library Manuals

If that still isn't enough detail, your next step is to read the Python library manual's entry for `sys` to get the full story. All of Python's standard manuals are available online, and they often install alongside Python itself. On Windows, the standard manuals are installed automatically, but here are a few simple pointers:

- On Windows, click the Start button, pick All Programs, select the Python entry there, and then choose the Python Manuals item. The manuals should magically appear on your display; as of Python 2.4, the manuals are provided as a Windows help file and so support searching and navigation.
- On Linux or Mac OS X, you may be able to click on the manuals' entries in a file explorer or start your browser from a shell command line and navigate to the library manual's HTML files on your machine.

- If you can't find the manuals on your computer, you can always read them online. Go to Python's website at *http://www.python.org* and follow the documentation links there. This website also has a simple searching utility for the manuals.

However you get started, be sure to pick the Library manual for things such as `sys`; this manual documents all of the standard library, built-in types and functions, and more. Python's standard manual set also includes a short tutorial, a language reference, extending references, and more.

Commercially Published References

At the risk of sounding like a marketing droid, I should mention that you can also purchase the Python manual set, printed and bound; see the book information page at *http://www.python.org* for details and links. Commercially published Python reference books are also available today, including *Python Essential Reference*, *Python in a Nutshell*, *Python Standard Library*, and *Python Pocket Reference*. Some of these books are more complete and come with examples, but the last one serves as a convenient memory jogger once you've taken a library tour or two.†

Introducing the sys Module

But enough about documentation sources (and scripting basics)—let's move on to system module details. As mentioned earlier, the `sys` and `os` modules form the core of much of Python's system-related tool set. To see how, we'll turn to a quick, interactive tour through some of the tools in these two modules before applying them in bigger examples. We'll start with `sys`, the smaller of the two; remember that to see a full list of all the attributes in `sys`, you need to pass it to the `dir` function (or see where we did so earlier in this chapter).

Platforms and Versions

Like most modules, `sys` includes both informational names and functions that take action. For instance, its attributes give us the name of the underlying operating system on which the platform code is running, the largest possible "natively sized" integer on this machine (though integers can be arbitrarily long in Python 3.X), and the version number of the Python interpreter running our code:

```
C:\...\PP4E\System> python
>>> import sys
```

† Full disclosure: I also wrote the last of the books listed as a replacement for the reference appendix that appeared in the first edition of this book; it's meant to be a supplement to the text you're reading, and its latest edition also serves as a translation resource for Python 2.X readers. As explained in the Preface, the book you're holding is meant as tutorial, not reference, so you'll probably want to find some sort of reference resource eventually (though I'm nearly narcissistic enough to require that it be mine).

```
>>> sys.platform, sys.maxsize, sys.version
('win32', 2147483647, '3.1.1 (r311:74483, Aug 17 2009, 17:02:12) ...more deleted...')

>>> if sys.platform[:3] == 'win': print('hello windows')
...
hello windows
```

If you have code that must act differently on different machines, simply test the `sys.platform` string as done here; although most of Python is cross-platform, nonportable tools are usually wrapped in `if` tests like the one here. For instance, we'll see later that some program launch and low-level console interaction tools may vary per platform—simply test `sys.platform` to pick the right tool for the machine on which your script is running.

The Module Search Path

The `sys` module also lets us inspect the module search path both interactively and within a Python program. `sys.path` is a list of directory name strings representing the true search path in a running Python interpreter. When a module is imported, Python scans this list from left to right, searching for the module's file on each directory named in the list. Because of that, this is the place to look to verify that your search path is really set as intended.‡

The `sys.path` list is simply initialized from your `PYTHONPATH` setting—the content of any *.pth* path files located in Python's directories on your machine plus system defaults—when the interpreter is first started up. In fact, if you inspect `sys.path` interactively, you'll notice quite a few directories that are not on your `PYTHONPATH`: `sys.path` also includes an indicator for the script's home directory (an empty string—something I'll explain in more detail after we meet `os.getcwd`) and a set of standard library directories that may vary per installation:

```
>>> sys.path
['', 'C:\\PP4thEd\\Examples', ...plus standard library paths deleted... ]
```

Surprisingly, `sys.path` can actually be *changed* by a program, too. A script can use list operations such as `append`, `extend`, `insert`, `pop`, and `remove`, as well as the `del` statement to configure the search path at runtime to include all the source directories to which it needs access. Python always uses the current `sys.path` setting to import, no matter what you've changed it to:

```
>>> sys.path.append(r'C:\mydir')
>>> sys.path
['', 'C:\\PP4thEd\\Examples', ...more deleted..., 'C:\\mydir']
```

‡ It's not impossible that Python sees `PYTHONPATH` differently than you do. A syntax error in your system shell configuration files may botch the setting of `PYTHONPATH`, even if it looks fine to you. On Windows, for example, if a space appears around the `=` of a DOS `set` command in your configuration file (e.g., `set NAME = VALUE`), you may actually set `NAME` to an empty string, not to `VALUE`!

Changing `sys.path` directly like this is an alternative to setting your `PYTHONPATH` shell variable, but not a very permanent one. Changes to `sys.path` are retained only until the Python process ends, and they must be remade every time you start a new Python program or session. However, some types of programs (e.g., scripts that run on a web server) may not be able to depend on `PYTHONPATH` settings; such scripts can instead configure `sys.path` on startup to include all the directories from which they will need to import modules. For a more concrete use case, see Example 1-34 in the prior chapter—there we had to tweak the search path dynamically this way, because the web server violated our import path assumptions.

Windows Directory Paths

Notice the use of a raw string literal in the `sys.path` configuration code: because backslashes normally introduce escape code sequences in Python strings, Windows users should be sure to either double up on backslashes when using them in DOS directory path strings (e.g., in `"C:\\dir"`, `\\` is an escape sequence that really means `\`), or use raw string constants to retain backslashes literally (e.g., `r"C:\dir"`).

If you inspect directory paths on Windows (as in the `sys.path` interaction listing), Python prints double `\\` to mean a single `\`. Technically, you can get away with a single `\` in a string if it is followed by a character Python does not recognize as the rest of an escape sequence, but doubles and raw strings are usually easier than memorizing escape code tables.

Also note that most Python library calls accept either forward (`/`) or backward (`\`) slashes as directory path separators, regardless of the underlying platform. That is, `/` usually works on Windows too and aids in making scripts portable to Unix. Tools in the `os` and `os.path` modules, described later in this chapter, further aid in script path portability.

The Loaded Modules Table

The `sys` module also contains hooks into the interpreter; `sys.modules`, for example, is a dictionary containing one *`name:module`* entry for every module imported in your Python session or program (really, in the calling Python process):

```
>>> sys.modules
{'reprlib': <module 'reprlib' from 'c:\python31\lib\reprlib.py'>, ...more deleted...

>>> list(sys.modules.keys())
 ['reprlib', 'heapq', '__future__', 'sre_compile', '_collections', 'locale', '_sre',
'functools', 'encodings', 'site', 'operator', 'io', '__main__', ...more deleted... ]

>>> sys
<module 'sys' (built-in)>
>>> sys.modules['sys']
<module 'sys' (built-in)>
```

We might use such a hook to write programs that display or otherwise process all the modules loaded by a program (just iterate over the keys of `sys.modules`).

Also in the interpret hooks category, an object's reference count is available via `sys.getrefcount`, and the names of modules built-in to the Python executable are listed in `sys.builtin_module_names`. See Python's library manual for details; these are mostly Python internals information, but such hooks can sometimes become important to programmers writing tools for other programmers to use.

Exception Details

Other attributes in the `sys` module allow us to fetch all the information related to the most recently raised Python exception. This is handy if we want to process exceptions in a more generic fashion. For instance, the `sys.exc_info` function returns a tuple with the latest exception's type, value, and traceback object. In the all class-based exception model that Python 3 uses, the first two of these correspond to the most recently raised exception's class, and the instance of it which was raised:

```
>>> try:
...     raise IndexError
... except:
...     print(sys.exc_info())
...
(<class 'IndexError'>, IndexError(), <traceback object at 0x019B8288>)
```

We might use such information to format our own error message to display in a GUI pop-up window or HTML web page (recall that by default, uncaught exceptions terminate programs with a Python error display). The first two items returned by this call have reasonable string displays when printed directly, and the third is a traceback object that can be processed with the standard `traceback` module:

```
>>> import traceback, sys
>>> def grail(x):
...     raise TypeError('already got one')
...
>>> try:
...     grail('arthur')
... except:
...     exc_info = sys.exc_info()
...     print(exc_info[0])
...     print(exc_info[1])
...     traceback.print_tb(exc_info[2])
...
<class 'TypeError'>
already got one
  File "<stdin>", line 2, in <module>
  File "<stdin>", line 2, in grail
```

The `traceback` module can also format messages as strings and route them to specific file objects; see the Python library manual for more details.

Other sys Module Exports

The `sys` module exports additional commonly-used tools that we will meet in the context of larger topics and examples introduced later in this part of the book. For instance:

- Command-line arguments show up as a list of strings called `sys.argv`.
- Standard streams are available as `sys.stdin`, `sys.stdout`, and `sys.stderr`.
- Program exit can be forced with `sys.exit` calls.

Since these lead us to bigger topics, though, we will cover them in sections of their own.

Introducing the os Module

As mentioned, `os` is the larger of the two core system modules. It contains all of the usual operating-system calls you use in C programs and shell scripts. Its calls deal with directories, processes, shell variables, and the like. Technically, this module provides POSIX tools—a portable standard for operating-system calls—along with platform-independent directory processing tools as the nested module `os.path`. Operationally, `os` serves as a largely portable interface to your computer's system calls: scripts written with `os` and `os.path` can usually be run unchanged on any platform. On some platforms, `os` includes extra tools available just for that platform (e.g., low-level process calls on Unix); by and large, though, it is as cross-platform as is technically feasible.

Tools in the os Module

Let's take a quick look at the basic interfaces in `os`. As a preview, Table 2-1 summarizes some of the most commonly used tools in the `os` module, organized by functional area.

Table 2-1. Commonly used os module tools

Tasks	Tools
Shell variables	`os.environ`
Running programs	`os.system`, `os.popen`, `os.execv`, `os.spawnv`
Spawning processes	`os.fork`, `os.pipe`, `os.waitpid`, `os.kill`
Descriptor files, locks	`os.open`, `os.read`, `os.write`
File processing	`os.remove`, `os.rename`, `os.mkfifo`, `os.mkdir`, `os.rmdir`
Administrative tools	`os.getcwd`, `os.chdir`, `os.chmod`, `os.getpid`, `os.listdir`, `os.access`
Portability tools	`os.sep`, `os.pathsep`, `os.curdir`, `os.path.split`, `os.path.join`
Pathname tools	`os.path.exists('path')`, `os.path.isdir('path')`, `os.path.getsize('path')`

If you inspect this module's attributes interactively, you get a huge list of names that will vary per Python release, will likely vary per platform, and isn't incredibly useful

until you've learned what each name means (I've let this line-wrap and removed most of this list to save space—run the command on your own):

```
>>> import os
>>> dir(os)
['F_OK', 'MutableMapping', 'O_APPEND', 'O_BINARY', 'O_CREAT', 'O_EXCL', 'O_NOINH
ERIT', 'O_RANDOM', 'O_RDONLY', 'O_RDWR', 'O_SEQUENTIAL', 'O_SHORT_LIVED', 'O_TEM
PORARY', 'O_TEXT', 'O_TRUNC', 'O_WRONLY', 'P_DETACH', 'P_NOWAIT', 'P_NOWAITO', '
P_OVERLAY', 'P_WAIT', 'R_OK', 'SEEK_CUR', 'SEEK_END', 'SEEK_SET', 'TMP_MAX',
...9 lines removed here...
'pardir', 'path', 'pathsep', 'pipe', 'popen', 'putenv', 'read', 'remove', 'rem
ovedirs', 'rename', 'renames', 'rmdir', 'sep', 'spawnl', 'spawnle', 'spawnv', 's
pawnve', 'startfile', 'stat', 'stat_float_times', 'stat_result', 'statvfs_result
', 'strerror', 'sys', 'system', 'times', 'umask', 'unlink', 'urandom', 'utime',
'waitpid', 'walk', 'write']
```

Besides all of these, the nested `os.path` module exports even more tools, most of which are related to processing file and directory names portably:

```
>>> dir(os.path)
['__all__', '__builtins__', '__doc__', '__file__', '__name__', '__package__',
'_get_altsep', '_get_bothseps', '_get_colon', '_get_dot', '_get_empty',
'_get_sep', '_getfullpathname', 'abspath', 'altsep', 'basename', 'commonprefix',
'curdir', 'defpath', 'devnull', 'dirname', 'exists', 'expanduser', 'expandvars',
'extsep', 'genericpath', 'getatime', 'getctime', 'getmtime', 'getsize', 'isabs',
'isdir', 'isfile', 'islink', 'ismount', 'join', 'lexists', 'normcase', 'normpath',
'os', 'pardir', 'pathsep', 'realpath', 'relpath', 'sep', 'split', 'splitdrive',
'splitext', 'splitunc', 'stat', 'supports_unicode_filenames', 'sys']
```

Administrative Tools

Just in case those massive listings aren't quite enough to go on, let's experiment interactively with some of the more commonly used `os` tools. Like `sys`, the `os` module comes with a collection of informational and administrative tools:

```
>>> os.getpid()
7980
>>> os.getcwd()
'C:\\PP4thEd\\Examples\\PP4E\\System'

>>> os.chdir(r'C:\Users')
>>> os.getcwd()
'C:\\Users'
```

As shown here, the `os.getpid` function gives the calling process's process ID (a unique system-defined identifier for a running program, useful for process control and unique name creation), and `os.getcwd` returns the current working directory. The current working directory is where files opened by your script are assumed to live, unless their names include explicit directory paths. That's why earlier I told you to run the following command in the directory where *more.py* lives:

```
C:\...\PP4E\System> python more.py more.py
```

The input filename argument here is given without an explicit directory path (though you could add one to page files in another directory). If you need to run in a different working directory, call the `os.chdir` function to change to a new directory; your code will run relative to the new directory for the rest of the program (or until the next `os.chdir` call). The next chapter will have more to say about the notion of a current working directory, and its relation to module imports when it explores script execution context.

Portability Constants

The `os` module also exports a set of names designed to make cross-platform programming simpler. The set includes platform-specific settings for path and directory separator characters, parent and current directory indicators, and the characters used to terminate lines on the underlying computer.

```
>>> os.pathsep, os.sep, os.pardir, os.curdir, os.linesep
(';', '\\', '..', '.', '\r\n')
```

`os.sep` is whatever character is used to separate directory components on the platform on which Python is running; it is automatically preset to `\` on Windows, `/` for POSIX machines, and `:` on some Macs. Similarly, `os.pathsep` provides the character that separates directories on directory lists, `:` for POSIX and `;` for DOS and Windows.

By using such attributes when composing and decomposing system-related strings in our scripts, we make the scripts fully portable. For instance, a call of the form `dir path.split(os.sep)` will correctly split platform-specific directory names into components, though `dirpath` may look like `dir\dir` on Windows, `dir/dir` on Linux, and `dir:dir` on some Macs. As mentioned, on Windows you can usually use forward slashes rather than backward slashes when giving filenames to be opened, but these portability constants allow scripts to be platform neutral in directory processing code.

Notice also how `os.linesep` comes back as `\r\n` here—the symbolic escape code which reflects the carriage-return + line-feed line terminator convention on Windows, which you don't normally notice when processing text files in Python. We'll learn more about end-of-line translations in Chapter 4.

Common os.path Tools

The nested module `os.path` provides a large set of directory-related tools of its own. For example, it includes portable functions for tasks such as checking a file's type (`isdir`, `isfile`, and others); testing file existence (`exists`); and fetching the size of a file by name (`getsize`):

```
>>> os.path.isdir(r'C:\Users'), os.path.isfile(r'C:\Users')
(True, False)
>>> os.path.isdir(r'C:\config.sys'), os.path.isfile(r'C:\config.sys')
(False, True)
>>> os.path.isdir('nonesuch'), os.path.isfile('nonesuch')
```

```
(False, False)

>>> os.path.exists(r'c:\Users\Brian')
False
>>> os.path.exists(r'c:\Users\Default')
True
>>> os.path.getsize(r'C:\autoexec.bat')
24
```

The `os.path.isdir` and `os.path.isfile` calls tell us whether a filename is a directory or a simple file; both return `False` if the named file does not exist (that is, nonexistence implies negation). We also get calls for splitting and joining directory path strings, which automatically use the directory name conventions on the platform on which Python is running:

```
>>> os.path.split(r'C:\temp\data.txt')
('C:\\temp', 'data.txt')

>>> os.path.join(r'C:\temp', 'output.txt')
'C:\\temp\\output.txt'

>>> name = r'C:\temp\data.txt'                          # Windows paths
>>> os.path.dirname(name), os.path.basename(name)
('C:\\temp', 'data.txt')

>>> name = '/home/lutz/temp/data.txt'                   # Unix-style paths
>>> os.path.dirname(name), os.path.basename(name)
('/home/lutz/temp', 'data.txt')

>>> os.path.splitext(r'C:\PP4thEd\Examples\PP4E\PyDemos.pyw')
('C:\\PP4thEd\\Examples\\PP4E\\PyDemos', '.pyw')
```

`os.path.split` separates a filename from its directory path, and `os.path.join` puts them back together—all in entirely portable fashion using the path conventions of the machine on which they are called. The `dirname` and `basename` calls here return the first and second items returned by a `split` simply as a convenience, and `splitext` strips the file extension (after the last `.`). Subtle point: it's almost equivalent to use string `split` and `join` method calls with the portable `os.sep` string, but not exactly:

```
>>> os.sep
'\\'
>>> pathname = r'C:\PP4thEd\Examples\PP4E\PyDemos.pyw'

>>> os.path.split(pathname)                             # split file from dir
('C:\\PP4thEd\\Examples\\PP4E', 'PyDemos.pyw')

>>> pathname.split(os.sep)                              # split on every slash
['C:', 'PP4thEd', 'Examples', 'PP4E', 'PyDemos.pyw']

>>> os.sep.join(pathname.split(os.sep))
'C:\\PP4thEd\\Examples\\PP4E\\PyDemos.pyw'

>>> os.path.join(*pathname.split(os.sep))
'C:PP4thEd\\Examples\\PP4E\\PyDemos.pyw'
```

The last join call require individual arguments (hence the *) but doesn't insert a first slash because of the Windows drive syntax; use the preceding `str.join` method instead if the difference matters. The `normpath` call comes in handy if your paths become a jumble of Unix and Windows separators:

```
>>> mixed
'C:\\temp\\public/files/index.html'
>>> os.path.normpath(mixed)
'C:\\temp\\public\\files\\index.html'
>>> print(os.path.normpath(r'C:\temp\\sub\.\file.ext'))
C:\temp\sub\file.ext
```

This module also has an `abspath` call that portably returns the full directory pathname of a file; it accounts for adding the current directory as a path prefix, `..` parent syntax, and more:

```
>>> os.chdir(r'C:\Users')
>>> os.getcwd()
'C:\\Users'
>>> os.path.abspath('')                      # empty string means the cwd
'C:\\Users'

>>> os.path.abspath('temp')                  # expand to full pathname in cwd
'C:\\Users\\temp'
>>> os.path.abspath(r'PP4E\dev')             # partial paths relative to cwd
'C:\\Users\\PP4E\\dev'

>>> os.path.abspath('.')                     # relative path syntax expanded
'C:\\Users'
>>> os.path.abspath('..')
'C:\\'
>>> os.path.abspath(r'..\examples')
'C:\\examples'

>>> os.path.abspath(r'C:\PP4thEd\chapters')  # absolute paths unchanged
'C:\\PP4thEd\\chapters'
>>> os.path.abspath(r'C:\temp\spam.txt')
'C:\\temp\\spam.txt'
```

Because filenames are relative to the current working directory when they aren't fully specified paths, the `os.path.abspath` function helps if you want to show users what directory is truly being used to store a file. On Windows, for example, when GUI-based programs are launched by clicking on file explorer icons and desktop shortcuts, the execution directory of the program is the clicked file's home directory, but that is not always obvious to the person doing the clicking; printing a file's `abspath` can help.

Running Shell Commands from Scripts

The `os` module is also the place where we run shell commands from within Python scripts. This concept is intertwined with others, such as streams, which we won't cover fully until the next chapter, but since this is a key concept employed throughout this

part of the book, let's take a quick first look at the basics here. Two `os` functions allow scripts to run any command line that you can type in a console window:

`os.system`
: Runs a shell command from a Python script

`os.popen`
: Runs a shell command and connects to its input or output streams

In addition, the relatively new `subprocess` module provides finer-grained control over streams of spawned shell commands and can be used as an alternative to, and even for the implementation of, the two calls above (albeit with some cost in extra code complexity).

What's a shell command?

To understand the scope of the calls listed above, we first need to define a few terms. In this text, the term *shell* means the system that reads and runs command-line strings on your computer, and *shell command* means a command-line string that you would normally enter at your computer's shell prompt.

For example, on Windows, you can start an MS-DOS console window (a.k.a. "Command Prompt") and type DOS commands there—commands such as `dir` to get a directory listing, `type` to view a file, names of programs you wish to start, and so on. DOS is the system shell, and commands such as `dir` and `type` are shell commands. On Linux and Mac OS X, you can start a new shell session by opening an xterm or terminal window and typing shell commands there too—`ls` to list directories, `cat` to view files, and so on. A variety of shells are available on Unix (e.g., csh, ksh), but they all read and run command lines. Here are two shell commands typed and run in an MS-DOS console box on Windows:

```
C:\...\PP4E\System> dir /B          ...type a shell command line
helloshell.py                       ...its output shows up here
more.py                             ...DOS is the shell on Windows
more.pyc
spam.txt
__init__.py

C:\...\PP4E\System> type helloshell.py
# a Python program
print('The Meaning of Life')
```

Running shell commands

None of this is directly related to Python, of course (despite the fact that Python command-line scripts are sometimes confusingly called "shell tools"). But because the `os` module's `system` and `popen` calls let Python scripts run any sort of command that the underlying system shell understands, our scripts can make use of every command-line tool available on the computer, whether it's coded in Python or not. For example, here

is some Python code that runs the two DOS shell commands typed at the shell prompt shown previously:

```
C:\...\PP4E\System> python
>>> import os
>>> os.system('dir /B')
helloshell.py
more.py
more.pyc
spam.txt
__init__.py
0
>>> os.system('type helloshell.py')
# a Python program
print('The Meaning of Life')
0

>>> os.system('type hellshell.py')
The system cannot find the file specified.
1
```

The 0s at the end of the first two commands here are just the return values of the system call itself (its exit status; zero generally means success). The system call can be used to run any command line that we could type at the shell's prompt (here, `C:\...\PP4E\System>`). The command's output normally shows up in the Python session's or program's standard output stream.

Communicating with shell commands

But what if we want to grab a command's output within a script? The `os.system` call simply runs a shell command line, but `os.popen` also connects to the standard input or output streams of the command; we get back a file-like object connected to the command's output by default (if we pass a `w` mode flag to `popen`, we connect to the command's input stream instead). By using this object to read the output of a command spawned with `popen`, we can intercept the text that would normally appear in the console window where a command line is typed:

```
>>> open('helloshell.py').read()
"# a Python program\nprint('The Meaning of Life')\n"

>>> text = os.popen('type helloshell.py').read()
>>> text
"# a Python program\nprint('The Meaning of Life')\n"

>>> listing = os.popen('dir /B').readlines()
>>> listing
['helloshell.py\n', 'more.py\n', 'more.pyc\n', 'spam.txt\n', '__init__.py\n']
```

Here, we first fetch a file's content the usual way (using Python files), then as the output of a shell `type` command. Reading the output of a `dir` command lets us get a listing of files in a directory that we can then process in a loop. We'll learn other ways to obtain such a list in Chapter 4; there we'll also learn how file *iterators* make the `readlines` call

in the os.popen example above unnecessary in most programs, except to display the list interactively as we did here (see also "subprocess, os.popen, and Iterators" on page 101 for more on the subject).

So far, we've run basic DOS commands; because these calls can run any command line that we can type at a shell prompt, they can also be used to launch other Python scripts. Assuming your system search path is set to locate your Python (so that you can use the shorter "python" in the following instead of the longer "C:\Python31\python"):

```
>>> os.system('python helloshell.py')       # run a Python program
The Meaning of Life
0
>>> output = os.popen('python helloshell.py').read()
>>> output
'The Meaning of Life\n'
```

In all of these examples, the command-line strings sent to system and popen are hard-coded, but there's no reason Python programs could not construct such strings at runtime using normal string operations (+, %, etc.). Given that commands can be dynamically built and run this way, system and popen turn Python scripts into flexible and portable tools for launching and orchestrating other programs. For example, a Python test "driver" script can be used to run programs coded in any language (e.g., C++, Java, Python) and analyze their output. We'll explore such a script in Chapter 6. We'll also revisit os.popen in the next chapter in conjunction with stream redirection; as we'll find, this call can also send *input* to programs.

The subprocess module alternative

As mentioned, in recent releases of Python the subprocess module can achieve the same effect as os.system and os.popen; it generally requires extra code but gives more control over how streams are connected and used. This becomes especially useful when streams are tied in more complex ways.

For example, to run a simple shell command like we did with os.system earlier, this new module's call function works roughly the same (running commands like "type" that are built into the shell on Windows requires extra protocol, though normal executables like "python" do not):

```
>>> import subprocess
>>> subprocess.call('python helloshell.py')             # roughly like os.system()
The Meaning of Life
0
>>> subprocess.call('cmd /C "type helloshell.py"')      # built-in shell cmd
# a Python program
print('The Meaning of Life')
0
>>> subprocess.call('type helloshell.py', shell=True)   # alternative for built-ins
# a Python program
print('The Meaning of Life')
0
```

Notice the `shell=True` in the last command here. This is a subtle and platform-dependent requirement:

- On Windows, we need to pass a `shell=True` argument to `subprocess` tools like `call` and `Popen` (shown ahead) in order to run commands built into the shell. Windows commands like "type" require this extra protocol, but normal executables like "python" do not.
- On Unix-like platforms, when `shell` is `False` (its default), the program command line is run directly by `os.execvp`, a call we'll meet in Chapter 5. If this argument is `True`, the command-line string is run through a shell instead, and you can specify the shell to use with additional arguments.

More on some of this later; for now, it's enough to note that you may need to pass `shell=True` to run some of the examples in this section and book in Unix-like environments, if they rely on shell features like program path lookup. Since I'm running code on Windows, this argument will often be omitted here.

Besides imitating `os.system`, we can similarly use this module to emulate the `os.popen` call used earlier, to run a shell command and obtain its standard output text in our script:

```
>>> pipe = subprocess.Popen('python helloshell.py', stdout=subprocess.PIPE)
>>> pipe.communicate()
(b'The Meaning of Life\r\n', None)
>>> pipe.returncode
0
```

Here, we connect the stdout stream to a pipe, and communicate to run the command to completion and receive its standard output and error streams' text; the command's exit status is available in an attribute after it completes. Alternatively, we can use other interfaces to read the command's standard output directly and wait for it to exit (which returns the exit status):

```
>>> pipe = subprocess.Popen('python helloshell.py', stdout=subprocess.PIPE)
>>> pipe.stdout.read()
b'The Meaning of Life\r\n'
>>> pipe.wait()
0
```

In fact, there are direct mappings from `os.popen` calls to `subprocess.Popen` objects:

```
>>> from subprocess import Popen, PIPE
>>> Popen('python helloshell.py', stdout=PIPE).communicate()[0]
b'The Meaning of Life\r\n'
>>>
>>> import os
>>> os.popen('python helloshell.py').read()
'The Meaning of Life\n'
```

As you can probably tell, `subprocess` is extra work in these relatively simple cases. It starts to look better, though, when we need to control additional streams in flexible ways. In fact, because it also allows us to process a command's error and input streams

in similar ways, in Python 3.X `subprocess` replaces the original `os.popen2`, `os.popen3`, and `os.popen4` calls which were available in Python 2.X; these are now just use cases for `subprocess` object interfaces. Because more advanced use cases for this module deal with standard streams, we'll postpone additional details about this module until we study stream redirection in the next chapter.

Shell command limitations

Before we move on, you should keep in mind two limitations of `system` and `popen`. First, although these two functions themselves are fairly portable, their use is really only as portable as the commands that they run. The preceding examples that run DOS `dir` and `type` shell commands, for instance, work only on Windows, and would have to be changed in order to run `ls` and `cat` commands on Unix-like platforms.

Second, it is important to remember that running Python files as programs this way is very different and generally much slower than importing program files and calling functions they define. When `os.system` and `os.popen` are called, they must start a brand-new, independent program running on your operating system (they generally run the command in a new process). When importing a program file as a module, the Python interpreter simply loads and runs the file's code in the same process in order to generate a module object. No other program is spawned along the way.§

There are good reasons to build systems as separate programs, too, and in the next chapter we'll explore things such as command-line arguments and streams that allow programs to pass information back and forth. But in many cases, imported modules are a faster and more direct way to compose systems.

If you plan to use these calls in earnest, you should also know that the `os.system` call normally blocks—that is, pauses—its caller until the spawned command line exits. On Linux and Unix-like platforms, the spawned command can generally be made to run independently and in parallel with the caller by adding an `&` shell background operator at the end of the command line:

```
os.system("python program.py arg arg &")
```

On Windows, spawning with a DOS `start` command will usually launch the command in parallel too:

```
os.system("start program.py arg arg")
```

§ The Python code `exec(open(file).read())` also runs a program file's code, but within the same process that called it. It's similar to an import in that regard, but it works more as if the file's text had been *pasted* into the calling program at the place where the `exec` call appears (unless explicit global or local namespace dictionaries are passed). Unlike imports, such an `exec` unconditionally reads and executes a file's code (it may be run more than once per process), no module object is generated by the file's execution, and unless optional namespace dictionaries are passed in, assignments in the file's code may overwrite variables in the scope where the `exec` appears; see other resources or the Python library manual for more details.

In fact, this is so useful that an `os.startfile` call was added in recent Python releases. This call opens a file with whatever program is listed in the Windows registry for the file's type—as though its icon has been clicked with the mouse cursor:

```
os.startfile("webpage.html")    # open file in your web browser
os.startfile("document.doc")    # open file in Microsoft Word
os.startfile("myscript.py")     # run file with Python
```

The `os.popen` call does not generally block its caller (by definition, the caller must be able to read or write the file object returned) but callers may still occasionally become blocked under both Windows and Linux if the pipe object is closed—e.g., when garbage is collected—before the spawned program exits or the pipe is read exhaustively (e.g., with its `read()` method). As we will see later in this part of the book, the Unix `os.fork/exec` and Windows `os.spawnv` calls can also be used to run parallel programs without blocking.

Because the `os` module's `system` and `popen` calls, as well as the `subprocess` module, also fall under the category of program launchers, stream redirectors, and cross-process communication devices, they will show up again in the following chapters, so we'll defer further details for the time being. If you're looking for more details right away, be sure to see the stream redirection section in the next chapter and the directory listings section in Chapter 4.

Other os Module Exports

That's as much of a tour around `os` as we have space for here. Since most other `os` module tools are even more difficult to appreciate outside the context of larger application topics, we'll postpone a deeper look at them until later chapters. But to let you sample the flavor of this module, here is a quick preview for reference. Among the `os` module's other weapons are these:

`os.environ`
: Fetches and sets shell environment variables

`os.fork`
: Spawns a new child process on Unix-like systems

`os.pipe`
: Communicates between programs

`os.execlp`
: Starts new programs

`os.spawnv`
: Starts new programs with lower-level control

`os.open`
: Opens a low-level descriptor-based file

`os.mkdir`
: Creates a new directory

`os.mkfifo`
: Creates a new named pipe

`os.stat`
: Fetches low-level file information

`os.remove`
: Deletes a file by its pathname

`os.walk`
: Applies a function or loop body to all parts of an entire directory tree

And so on. One caution up front: the `os` module provides a set of file `open`, `read`, and `write` calls, but all of these deal with low-level file access and are entirely distinct from Python's built-in `stdio` file objects that we create with the built-in `open` function. You should normally use the built-in `open` function, not the `os` module, for all but very special file-processing needs (e.g., opening with exclusive access file locking).

In the next chapter we will apply `sys` and `os` tools such as those we've introduced here to implement common system-level tasks, but this book doesn't have space to provide an exhaustive list of the contents of modules we will meet along the way. Again, if you have not already done so, you should become acquainted with the contents of modules such as `os` and `sys` using the resources described earlier. For now, let's move on to explore additional system tools in the context of broader system programming concepts—the context surrounding a running script.

subprocess, os.popen, and Iterators

In Chapter 4, we'll explore file iterators, but you've probably already studied the basics prior to picking up this book. Because `os.popen` objects have an iterator that reads one line at a time, their `readlines` method call is usually superfluous. For example, the following steps through lines produced by another program without any explicit reads:

```
>>> import os
>>> for line in os.popen('dir /B *.py'): print(line, end='')
...
helloshell.py
more.py
__init__.py
```

Interestingly, Python 3.1 implements `os.popen` using the `subprocess.Popen` object that we studied in this chapter. You can see this for yourself in file *os.py* in the Python standard library on your machine (see *C:\Python31\Lib* on Windows); the `os.popen` result is an object that manages the `Popen` object and its piped stream:

```
>>> I = os.popen('dir /B *.py')
>>> I
<os._wrap_close object at 0x013BC750>
```

Because this pipe wrapper object defines an `__iter__` method, it supports line iteration, both automatic (e.g., the `for` loop above) and manual. Curiously, although the pipe wrapper object supports direct `__next__` method calls as though it were its own iterator

(just like simple files), it does not support the `next` built-in function, even though the latter is supposed to simply call the former:

```
>>> I = os.popen('dir /B *.py')
>>> I.__next__()
'helloshell.py\n'

>>> I = os.popen('dir /B *.py')
>>> next(I)
TypeError: _wrap_close object is not an iterator
```

The reason for this is subtle—direct `__next__` calls are intercepted by a `__getattr__` defined in the pipe wrapper object, and are properly delegated to the wrapped object; but `next` function calls invoke Python's operator overloading machinery, which in 3.X bypasses the wrapper's `__getattr__` for special method names like `__next__`. Since the pipe wrapper object doesn't define a `__next__` of its own, the call is not caught and delegated, and the `next` built-in fails. As explained in full in the book *Learning Python*, the wrapper's `__getattr__` isn't tried because 3.X begins such searches at the class, not the instance.

This behavior may or may not have been anticipated, and you don't need to care if you iterate over pipe lines automatically with `for` loops, comprehensions, and other tools. To code manual iterations robustly, though, be sure to call the `iter` built-in first—this invokes the `__iter__` defined in the pipe wrapper object itself, to correctly support both flavors of advancement:

```
>>> I = os.popen('dir /B *.py')
>>> I = iter(I)                      # what for loops do
>>> I.__next__()                     # now both forms work
'helloshell.py\n'
>>> next(I)
'more.py\n'
```

CHAPTER 3

Script Execution Context

"I'd Like to Have an Argument, Please"

Python scripts don't run in a vacuum (despite what you may have heard). Depending on platforms and startup procedures, Python programs may have all sorts of enclosing context—information automatically passed in to the program by the operating system when the program starts up. For instance, scripts have access to the following sorts of system-level inputs and interfaces:

Current working directory
: `os.getcwd` gives access to the directory from which a script is started, and many file tools use its value implicitly.

Command-line arguments
: `sys.argv` gives access to words typed on the command line that are used to start the program and that serve as script inputs.

Shell variables
: `os.environ` provides an interface to names assigned in the enclosing shell (or a parent program) and passed in to the script.

Standard streams
: `sys.stdin`, `stdout`, and `stderr` export the three input/output streams that are at the heart of command-line shell tools, and can be leveraged by scripts with `print` options, the `os.popen` call and `subprocess` module introduced in Chapter 2, the `io.StringIO` class, and more.

Such tools can serve as inputs to scripts, configuration parameters, and so on. In this chapter, we will explore all these four context's tools—both their Python interfaces and their typical roles.

Current Working Directory

The notion of the current working directory (CWD) turns out to be a key concept in some scripts' execution: it's always the implicit place where files processed by the script are assumed to reside unless their names have absolute directory paths. As we saw earlier, `os.getcwd` lets a script fetch the CWD name explicitly, and `os.chdir` allows a script to move to a new CWD.

Keep in mind, though, that filenames without full pathnames map to the CWD and have nothing to do with your `PYTHONPATH` setting. Technically, a script is always launched from the CWD, not the directory containing the script file. Conversely, imports always first search the directory containing the script, not the CWD (unless the script happens to also be located in the CWD). Since this distinction is subtle and tends to trip up beginners, let's explore it in a bit more detail.

CWD, Files, and Import Paths

When you run a Python script by typing a shell command line such as `python dir1\dir2\file.py`, the CWD is the directory you were in when you typed this command, not *dir1\dir2*. On the other hand, Python automatically adds the identity of the script's home directory to the front of the module search path such that *file.py* can always import other files in *dir1\dir2* no matter where it is run from. To illustrate, let's write a simple script to echo both its CWD and its module search path:

```
C:\...\PP4E\System> type whereami.py
import os, sys
print('my os.getcwd =>', os.getcwd())          # show my cwd execution dir
print('my sys.path  =>', sys.path[:6])         # show first 6 import paths
input()                                        # wait for keypress if clicked
```

Now, running this script in the directory in which it resides sets the CWD as expected and adds it to the front of the module import search path. We met the `sys.path` module search path earlier; its first entry might also be the empty string to designate CWD when you're working interactively, and most of the CWD has been truncated to "..." here for display:

```
C:\...\PP4E\System> set PYTHONPATH=C:\PP4thEd\Examples
C:\...\PP4E\System> python whereami.py
my os.getcwd => C:\...\PP4E\System
my sys.path  => ['C:\\...\\PP4E\\System', 'C:\\PP4thEd\\Examples', ...more... ]
```

But if we run this script from other places, the CWD moves with us (it's the directory where we type commands), and Python adds a directory to the front of the module search path that allows the script to still see files in its own home directory. For instance, when running from one level up (`..`), the `System` name added to the front of `sys.path` will be the first directory that Python searches for imports within *whereami.py*; it points imports back to the directory containing the script that was run. Filenames without

complete paths, though, will be mapped to the CWD (*C:\PP4thEd\Examples\PP4E*), not the *System* subdirectory nested there:

```
C:\...\PP4E\System> cd ..
C:\...\PP4E> python System\whereami.py
my os.getcwd => C:\...\PP4E
my sys.path  => ['C:\\...\\PP4E\\System', 'C:\\PP4thEd\\Examples', ...more... ]

C:\...\PP4E> cd System\temp
C:\...\PP4E\System\temp> python ..\whereami.py
my os.getcwd => C:\...\PP4E\System\temp
my sys.path  => ['C:\\...\\PP4E\\System', 'C:\\PP4thEd\\Examples', ...]
```

The net effect is that filenames without directory paths in a script will be mapped to the place where the *command* was typed (`os.getcwd`), but imports still have access to the directory of the *script* being run (via the front of `sys.path`). Finally, when a file is launched by clicking its icon, the CWD is just the directory that contains the clicked file. The following output, for example, appears in a new DOS console box when *whereami.py* is double-clicked in Windows Explorer:

```
my os.getcwd => C:\...\PP4E\System
my sys.path  => ['C:\\...\\PP4E\\System', ...more... ]
```

In this case, both the CWD used for filenames and the first import search directory are the directory containing the script file. This all usually works out just as you expect, but there are two pitfalls to avoid:

- Filenames might need to include complete directory paths if scripts cannot be sure from where they will be run.
- Command-line scripts cannot always rely on the CWD to gain import visibility to files that are not in their own directories; instead, use `PYTHONPATH` settings and package import paths to access modules in other directories.

For example, scripts in this book, regardless of how they are run, can always import other files in their own home directories without package path imports (`import filehere`), but must go through the PP4E package root to find files anywhere else in the examples tree (`from PP4E.dir1.dir2 import filethere`), even if they are run from the directory containing the desired external module. As usual for modules, the *PP4E\dir1\dir2* directory name could also be added to `PYTHONPATH` to make files there visible everywhere without package path imports (though adding more directories to `PYTHONPATH` increases the likelihood of name clashes). In either case, though, imports are always resolved to the script's home directory or other Python search path settings, not to the CWD.

CWD and Command Lines

This distinction between the CWD and import search paths explains why many scripts in this book designed to operate in the current working directory (instead of one whose name is passed in) are run with command lines such as this one:

```
C:\temp> python C:\...\PP4E\Tools\cleanpyc.py                process cwd
```

In this example, the Python script file itself lives in the directory *C:\...\PP4E\Tools*, but because it is run from *C:\temp*, it processes the files located in *C:\temp* (i.e., in the CWD, not in the script's home directory). To process files elsewhere with such a script, simply `cd` to the directory to be processed to change the CWD:

```
C:\temp> cd C:\PP4thEd\Examples
C:\PP4thEd\Examples> python C:\...\PP4E\Tools\cleanpyc.py      process cwd
```

Because the CWD is always implied, a `cd` command tells the script which directory to process in no less certain terms than passing a directory name to the script explicitly, like this (portability note: you may need to add quotes around the `*.py` in this and other command-line examples to prevent it from being expanded in some Unix shells):

```
C:\...\PP4E\Tools> python find.py *.py C:\temp               process named dir
```

In this command line, the CWD is the directory containing the script to be run (notice that the script filename has no directory path prefix); but since this script processes a directory named explicitly on the command line (*C:\temp*), the CWD is irrelevant. Finally, if we want to run such a script located in some other directory in order to process files located in yet another directory, we can simply give directory paths to both:

```
C:\temp> python C:\...\PP4E\Tools\find.py *.cxx C:\PP4thEd\Examples\PP4E
```

Here, the script has import visibility to files in its *PP4E\Tools* home directory and processes files in the directory named on the command line, but the CWD is something else entirely (*C:\temp*). This last form is more to type, of course, but watch for a variety of CWD and explicit script-path command lines like these in this book.

Command-Line Arguments

The `sys` module is also where Python makes available the words typed on the command that is used to start a Python script. These words are usually referred to as command-line arguments and show up in `sys.argv`, a built-in list of strings. C programmers may notice its similarity to the C `argv` array (an array of C strings). It's not much to look at interactively, because no command-line arguments are passed to start up Python in this mode:

```
>>> import sys
>>> sys.argv
['']
```

To really see what arguments are about, we need to run a script from the shell command line. Example 3-1 shows an unreasonably simple one that just prints the `argv` list for inspection.

Example 3-1. PP4E\System\testargv.py

```
import sys
print(sys.argv)
```

Running this script prints the command-line arguments list; note that the first item is always the name of the executed Python script file itself, no matter how the script was started (see "Executable Scripts on Unix" on page 108).

```
C:\...\PP4E\System> python testargv.py
['testargv.py']

C:\...\PP4E\System> python testargv.py spam eggs cheese
['testargv.py', 'spam', 'eggs', 'cheese']

C:\...\PP4E\System> python testargv.py -i data.txt -o results.txt
['testargv.py', '-i', 'data.txt', '-o', 'results.txt']
```

The last command here illustrates a common convention. Much like function arguments, command-line options are sometimes passed by position and sometimes by name using a "-name value" word pair. For instance, the pair `-i data.txt` means the `-i` option's value is `data.txt` (e.g., an input filename). Any words can be listed, but programs usually impose some sort of structure on them.

Command-line arguments play the same role in programs that function arguments do in functions: they are simply a way to pass information to a program that can vary per program run. Because they don't have to be hardcoded, they allow scripts to be more generally useful. For example, a file-processing script can use a command-line argument as the name of the file it should process; see Chapter 2's *more.py* script (Example 2-1) for a prime example. Other scripts might accept processing mode flags, Internet addresses, and so on.

Parsing Command-Line Arguments

Once you start using command-line arguments regularly, though, you'll probably find it inconvenient to keep writing code that fishes through the list looking for words. More typically, programs translate the arguments list on startup into structures that are more conveniently processed. Here's one way to do it: the script in Example 3-2 scans the `argv` list looking for `-optionname optionvalue` word pairs and stuffs them into a dictionary by option name for easy retrieval.

Example 3-2. PP4E\System\testargv2.py

```
"collect command-line options in a dictionary"

def getopts(argv):
    opts = {}
    while argv:
        if argv[0][0] == '-':                  # find "-name value" pairs
            opts[argv[0]] = argv[1]            # dict key is "-name" arg
            argv = argv[2:]
        else:
            argv = argv[1:]
    return opts

if __name__ == '__main__':
    from sys import argv                       # example client code
    myargs = getopts(argv)
    if '-i' in myargs:
        print(myargs['-i'])
    print(myargs)
```

You might import and use such a function in all your command-line tools. When run by itself, this file just prints the formatted argument dictionary:

```
C:\...\PP4E\System> python testargv2.py
{}

C:\...\PP4E\System> python testargv2.py -i data.txt -o results.txt
data.txt
{'-o': 'results.txt', '-i': 'data.txt'}
```

Naturally, we could get much more sophisticated here in terms of argument patterns, error checking, and the like. For more complex command lines, we could also use command-line processing tools in the Python standard library to parse arguments:

- The `getopt` module, modeled after a Unix/C utility of the same name
- The `optparse` module, a newer alternative, generally considered to be more powerful

Both of these are documented in Python's library manual, which also provides usage examples which we'll defer to here in the interest of space. In general, the more configurable your scripts, the more you must invest in command-line processing logic complexity.

Executable Scripts on Unix

Unix and Linux users: you can also make text files of Python source code directly executable by adding a special line at the top with the path to the Python interpreter and giving the file executable permission. For instance, type this code into a text file called *myscript*:

```
#!/usr/bin/python
print('And nice red uniforms')
```

The first line is normally taken as a comment by Python (it starts with a `#`); but when this file is run, the operating system sends lines in this file to the interpreter listed after `#!` in line 1. If this file is made directly executable with a shell command of the form `chmod +x myscript`, it can be run directly without typing `python` in the command, as though it were a binary executable program:

```
% myscript a b c
And nice red uniforms
```

When run this way, `sys.argv` will still have the script's name as the first word in the list: `["myscript", "a", "b", "c"]`, exactly as if the script had been run with the more explicit and portable command form `python myscript a b c`. Making scripts directly executable is actually a Unix trick, not a Python feature, but it's worth pointing out that it can be made a bit less machine dependent by listing the Unix `env` command at the top instead of a hardcoded path to the Python executable:

```
#!/usr/bin/env python
print('Wait for it...')
```

When coded this way, the operating system will employ your environment variable settings to locate your Python interpreter (your `PATH` variable, on most platforms). If you run the same script on many machines, you need only change your environment settings on each machine (you don't need to edit Python script code). Of course, you can always run Python files with a more explicit command line:

```
% python myscript a b c
```

This assumes that the `python` interpreter program is on your system's search path setting (otherwise, you need to type its full path), but it works on any Python platform with a command line. Since this is more portable, I generally use this convention in the book's examples, but consult your Unix manpages for more details on any of the topics mentioned here. Even so, these special `#!` lines will show up in many examples in this book just in case readers want to run them as executables on Unix or Linux; on other platforms, they are simply ignored as Python comments.

Note that on recent flavors of Windows, you can usually also type a script's filename directly (without the word *python*) to make it go, and you don't have to add a `#!` line at the top. Python uses the Windows registry on this platform to declare itself as the program that opens files with Python extensions (*.py* and others). This is also why you can launch files on Windows by clicking on them.

Shell Environment Variables

Shell variables, sometimes known as environment variables, are made available to Python scripts as `os.environ`, a Python dictionary-like object with one entry per variable setting in the shell. Shell variables live outside the Python system; they are often set at your system prompt or within startup files or control-panel GUIs and typically serve as system-wide configuration inputs to programs.

In fact, by now you should be familiar with a prime example: the PYTHONPATH module search path setting is a shell variable used by Python to import modules. By setting it once in your operating system, its value is available every time a Python program is run. Shell variables can also be set by programs to serve as inputs to other programs in an application; because their values are normally inherited by spawned programs, they can be used as a simple form of interprocess communication.

Fetching Shell Variables

In Python, the surrounding shell environment becomes a simple preset object, not special syntax. Indexing `os.environ` by the desired shell variable's name string (e.g., `os.environ['USER']`) is the moral equivalent of adding a dollar sign before a variable name in most Unix shells (e.g., `$USER`), using surrounding percent signs on DOS (`%USER%`), and calling `getenv("USER")` in a C program. Let's start up an interactive session to experiment (run in Python 3.1 on a Windows 7 laptop):

```
>>> import os
>>> os.environ.keys()
KeysView(<os._Environ object at 0x013B8C70>)

>>> list(os.environ.keys())
['TMP', 'COMPUTERNAME', 'USERDOMAIN', 'PSMODULEPATH', 'COMMONPROGRAMFILES',
...many more deleted...
'NUMBER_OF_PROCESSORS', 'PROCESSOR_LEVEL', 'USERPROFILE', 'OS', 'PUBLIC', 'QTJAVA']

>>> os.environ['TEMP']
'C:\\Users\\mark\\AppData\\Local\\Temp'
```

Here, the `keys` method returns an iterable of assigned variables, and indexing fetches the value of the shell variable `TEMP` on Windows. This works the same way on Linux, but other variables are generally preset when Python starts up. Since we know about `PYTHONPATH`, let's peek at its setting within Python to verify its content (as I wrote this, mine was set to the root of the book examples tree for this fourth edition, as well as a temporary development location):

```
>>> os.environ['PYTHONPATH']
'C:\\PP4thEd\\Examples;C:\\Users\\Mark\\temp'

>>> for srcdir in os.environ['PYTHONPATH'].split(os.pathsep):
...     print(srcdir)
...
C:\PP4thEd\Examples
C:\Users\Mark\temp

>>> import sys
>>> sys.path[:3]
['', 'C:\\PP4thEd\\Examples', 'C:\\Users\\Mark\\temp']
```

`PYTHONPATH` is a string of directory paths separated by whatever character is used to separate items in such paths on your platform (e.g., `;` on DOS/Windows, `:` on Unix and Linux). To split it into its components, we pass to the `split` string method an

`os.pathsep` delimiter—a portable setting that gives the proper separator for the underlying machine. As usual, `sys.path` is the actual search path at runtime, and reflects the result of merging in the `PYTHONPATH` setting after the current directory.

Changing Shell Variables

Like normal dictionaries, the `os.environ` object supports both key indexing and *assignment*. As for dictionaries, assignments change the value of the key:

```
>>> os.environ['TEMP']
'C:\\Users\\mark\\AppData\\Local\\Temp
>>> os.environ['TEMP'] = r'c:\temp'
>>> os.environ['TEMP']
'c:\\temp'
```

But something extra happens here. In all recent Python releases, values assigned to `os.environ` keys in this fashion are automatically *exported* to other parts of the application. That is, key assignments change both the `os.environ` object in the Python program as well as the associated variable in the enclosing *shell* environment of the running program's process. Its new value becomes visible to the Python program, all linked-in C modules, and any programs spawned by the Python process.

Internally, key assignments to `os.environ` call `os.putenv`—a function that changes the shell variable outside the boundaries of the Python interpreter. To demonstrate how this works, we need a couple of scripts that set and fetch shell variables; the first is shown in Example 3-3.

Example 3-3. PP4E\System\Environment\setenv.py

```
import os
print('setenv...', end=' ')
print(os.environ['USER'])               # show current shell variable value

os.environ['USER'] = 'Brian'            # runs os.putenv behind the scenes
os.system('python echoenv.py')

os.environ['USER'] = 'Arthur'           # changes passed to spawned programs
os.system('python echoenv.py')          # and linked-in C library modules

os.environ['USER'] = input('?')
print(os.popen('python echoenv.py').read())
```

This *setenv.py* script simply changes a shell variable, `USER`, and spawns another script that echoes this variable's value, as shown in Example 3-4.

Example 3-4. PP4E\System\Environment\echoenv.py

```
import os
print('echoenv...', end=' ')
print('Hello,', os.environ['USER'])
```

No matter how we run *echoenv.py*, it displays the value of USER in the enclosing shell; when run from the command line, this value comes from whatever we've set the variable to in the shell itself:

```
C:\...\PP4E\System\Environment> set USER=Bob

C:\...\PP4E\System\Environment> python echoenv.py
echoenv... Hello, Bob
```

When spawned by another script such as *setenv.py* using the os.system and os.popen tools we met earlier, though, *echoenv.py* gets whatever USER settings its parent program has made:

```
C:\...\PP4E\System\Environment> python setenv.py
setenv... Bob
echoenv... Hello, Brian
echoenv... Hello, Arthur
?Gumby
echoenv... Hello, Gumby

C:\...\PP4E\System\Environment> echo %USER%
Bob
```

This works the same way on Linux. In general terms, a spawned program always *inherits* environment settings from its parents. *Spawned* programs are programs started with Python tools such as os.spawnv, the os.fork/exec combination on Unix-like platforms, and os.popen, os.system, and the subprocess module on a variety of platforms. All programs thus launched get the environment variable settings that exist in the parent at launch time.*

From a larger perspective, setting shell variables like this before starting a new program is one way to pass information into the new program. For instance, a Python configuration script might tailor the PYTHONPATH variable to include custom directories just before launching another Python script; the launched script will have the custom search path in its sys.path because shell variables are passed down to children (in fact, watch for such a launcher script to appear at the end of Chapter 6).

Shell Variable Fine Points: Parents, putenv, and getenv

Notice the last command in the preceding example—the USER variable is back to its original value after the top-level Python program exits. Assignments to os.environ keys are passed outside the interpreter and *down* the spawned programs chain, but never back *up* to parent program processes (including the system shell). This is also true in C programs that use the putenv library call, and it isn't a Python limitation per se.

* This is by default. Some program-launching tools also let scripts pass environment settings that are different from their own to child programs. For instance, the os.spawnve call is like os.spawnv, but it accepts a dictionary argument representing the shell environment to be passed to the started program. Some os.exec* variants (ones with an "e" at the end of their names) similarly accept explicit environments; see the os.exec* call formats in Chapter 5 for more details.

It's also likely to be a nonissue if a Python script is at the top of your application. But keep in mind that shell settings made within a program usually endure only for that program's run and for the run of its spawned children. If you need to export a shell variable setting so that it lives on after Python exits, you may be able to find platform-specific extensions that do this; search *http://www.python.org* or the Web at large.

Another subtlety: as implemented today, changes to `os.environ` automatically call `os.putenv`, which runs the `putenv` call in the C library if it is available on your platform to export the setting outside Python to any linked-in C code. However, although `os.environ` changes call `os.putenv`, direct calls to `os.putenv` do not update `os.environ` to reflect the change. Because of this, the `os.environ` mapping interface is generally preferred to `os.putenv`.

Also note that environment settings are loaded into `os.environ` on startup and not on each fetch; hence, changes made by linked-in C code after startup may not be reflected in `os.environ`. Python does have a more focused `os.getenv` call today, but it is simply translated into an `os.environ` fetch on most platforms (or all, in 3.X), not into a call to `getenv` in the C library. Most applications won't need to care, especially if they are pure Python code. On platforms without a `putenv` call, `os.environ` can be passed as a parameter to program startup tools to set the spawned program's environment.

Standard Streams

The `sys` module is also the place where the standard input, output, and error streams of your Python programs live; these turn out to be another common way for programs to communicate:

```
>>> import sys
>>> for f in (sys.stdin, sys.stdout, sys.stderr): print(f)
...
<_io.TextIOWrapper name='<stdin>' encoding='cp437'>
<_io.TextIOWrapper name='<stdout>' encoding='cp437'>
<_io.TextIOWrapper name='<stderr>' encoding='cp437'>
```

The standard streams are simply preopened Python file objects that are automatically connected to your program's standard streams when Python starts up. By default, all of them are tied to the console window where Python (or a Python program) was started. Because the `print` and `input` built-in functions are really nothing more than user-friendly interfaces to the standard output and input streams, they are similar to using `stdout` and `stdin` in `sys` directly:

```
>>> print('hello stdout world')
hello stdout world

>>> sys.stdout.write('hello stdout world' + '\n')
hello stdout world
19
```

```
>>> input('hello stdin world>')
hello stdin world>spam
'spam'

>>> print('hello stdin world>'); sys.stdin.readline()[:-1]
hello stdin world>
eggs
'eggs'
```

Standard Streams on Windows

Windows users: if you click a *.py* Python program's filename in a Windows file explorer to start it (or launch it with `os.system`), a DOS console window automatically pops up to serve as the program's standard stream. If your program makes windows of its own, you can avoid this console pop-up window by naming your program's source-code file with a *.pyw* extension, not with a *.py* extension. The *.pyw* extension simply means a *.py* source file without a DOS pop up on Windows (it uses Windows registry settings to run a custom version of Python). A *.pyw* file may also be imported as usual.

Also note that because printed output goes to this DOS pop up when a program is clicked, scripts that simply print text and exit will generate an odd "flash"—the DOS console box pops up, output is printed into it, and the pop up goes away immediately (not the most user-friendly of features!). To keep the DOS pop-up box around so that you can read printed output, simply add an `input()` call at the bottom of your script to pause for an Enter key press before exiting.

Redirecting Streams to Files and Programs

Technically, standard output (and `print`) text appears in the console window where a program was started, standard input (and `input`) text comes from the keyboard, and standard error text is used to print Python error messages to the console window. At least that's the default. It's also possible to *redirect* these streams both to files and to other programs at the system shell, as well as to arbitrary objects within a Python script. On most systems, such redirections make it easy to reuse and combine general-purpose command-line utilities.

Redirection is useful for things like canned (precoded) test inputs: we can apply a single test script to any set of inputs by simply redirecting the standard input stream to a different file each time the script is run. Similarly, redirecting the standard output stream lets us save and later analyze a program's output; for example, testing systems might compare the saved standard output of a script with a file of expected output to detect failures.

Although it's a powerful paradigm, redirection turns out to be straightforward to use. For instance, consider the simple read-evaluate-print loop program in Example 3-5.

Example 3-5. PP4E\System\Streams\teststreams.py

```
"read numbers till eof and show squares"

def interact():
    print('Hello stream world')                    # print sends to sys.stdout
    while True:
        try:
            reply = input('Enter a number>')       # input reads sys.stdin
        except EOFError:
            break                                  # raises an except on eof
        else:                                      # input given as a string
            num = int(reply)
            print("%d squared is %d" % (num, num ** 2))
    print('Bye')

if __name__ == '__main__':
    interact()                                     # when run, not imported
```

As usual, the `interact` function here is automatically executed when this file is run, not when it is imported. By default, running this file from a system command line makes that standard stream appear where you typed the Python command. The script simply reads numbers until it reaches end-of-file in the standard input stream (on Windows, end-of-file is usually the two-key combination Ctrl-Z; on Unix, type Ctrl-D instead†):

```
C:\...\PP4E\System\Streams> python teststreams.py
Hello stream world
Enter a number>12
12 squared is 144
Enter a number>10
10 squared is 100
Enter a number>^Z
Bye
```

But on both Windows and Unix-like platforms, we can redirect the standard input stream to come from a file with the `< filename` shell syntax. Here is a command session in a DOS console box on Windows that forces the script to read its input from a text file, *input.txt*. It's the same on Linux, but replace the DOS `type` command with a Unix `cat` command:

```
C:\...\PP4E\System\Streams> type input.txt
8
6

C:\...\PP4E\System\Streams> python teststreams.py < input.txt
Hello stream world
```

† Notice that `input` raises an exception to signal end-of-file, but file read methods simply return an empty string for this condition. Because `input` also strips the end-of-line character at the end of lines, an empty string result means an empty line, so an exception is necessary to specify the end-of-file condition. File read methods retain the end-of-line character and denote an empty line as `"\n"` instead of `""`. This is one way in which reading `sys.stdin` directly differs from `input`. The latter also accepts a prompt string that is automatically printed before input is accepted.

```
Enter a number>8 squared is 64
Enter a number>6 squared is 36
Enter a number>Bye
```

Here, the *input.txt* file automates the input we would normally type interactively—the script reads from this file rather than from the keyboard. Standard output can be similarly redirected to go to a file with the `> filename` shell syntax. In fact, we can combine input and output redirection in a single command:

```
C:\...\PP4E\System\Streams> python teststreams.py < input.txt > output.txt

C:\...\PP4E\System\Streams> type output.txt
Hello stream world
Enter a number>8 squared is 64
Enter a number>6 squared is 36
Enter a number>Bye
```

This time, the Python script's input and output are both mapped to text files, not to the interactive console session.

Chaining programs with pipes

On Windows and Unix-like platforms, it's also possible to send the standard output of one program to the standard input of another using the | shell character between two commands. This is usually called a "pipe" operation because the shell creates a pipeline that connects the output and input of two commands. Let's send the output of the Python script to the standard `more` command-line program's input to see how this works:

```
C:\...\PP4E\System\Streams> python teststreams.py < input.txt | more

Hello stream world
Enter a number>8 squared is 64
Enter a number>6 squared is 36
Enter a number>Bye
```

Here, `teststreams`'s standard input comes from a file again, but its output (written by `print` calls) is sent to another program, not to a file or window. The receiving program is `more`, a standard command-line paging program available on Windows and Unix-like platforms. Because Python ties scripts into the standard stream model, though, Python scripts can be used on both ends. One Python script's output can always be piped into another Python script's input:

```
C:\...\PP4E\System\Streams> type writer.py
print("Help! Help! I'm being repressed!")
print(42)

C:\...\PP4E\System\Streams> type reader.py
print('Got this: "%s"' % input())
import sys
data = sys.stdin.readline()[:-1]
print('The meaning of life is', data, int(data) * 2)
```

```
C:\...\PP4E\System\Streams> python writer.py
Help! Help! I'm being repressed!
42

C:\...\PP4E\System\Streams> python writer.py | python reader.py
Got this: "Help! Help! I'm being repressed!"
The meaning of life is 42 84
```

This time, two Python programs are connected. Script `reader` gets input from script `writer`; both scripts simply read and write, oblivious to stream mechanics. In practice, such chaining of programs is a simple form of cross-program communications. It makes it easy to *reuse* utilities written to communicate via `stdin` and `stdout` in ways we never anticipated. For instance, a Python program that sorts `stdin` text could be applied to any data source we like, including the output of other scripts. Consider the Python command-line utility scripts in Examples 3-6 and 3-7 which sort and sum lines in the standard input stream.

Example 3-6. PP4E\System\Streams\sorter.py

```
import sys                                  # or sorted(sys.stdin)
lines = sys.stdin.readlines()               # sort stdin input lines,
lines.sort()                                # send result to stdout
for line in lines: print(line, end='')      # for further processing
```

Example 3-7. PP4E\System\Streams\adder.py

```
import sys
sum = 0
while True:
    try:
        line = input()                  # or call sys.stdin.readlines()
    except EOFError:                    # or for line in sys.stdin:
        break                           # input strips \n at end
    else:
        sum += int(line)                # was sting.atoi() in 2nd ed
print(sum)
```

We can apply such general-purpose tools in a variety of ways at the shell command line to sort and sum arbitrary files and program outputs (Windows note: on my prior XP machine and Python 2.X, I had to type "python file.py" here, not just "file.py," or else the input redirection failed; with Python 3.X on Windows 7 today, either form works):

```
C:\...\PP4E\System\Streams> type data.txt
123
000
999
042

C:\...\PP4E\System\Streams> python sorter.py < data.txt          sort a file
000
042
123
999
```

```
C:\...\PP4E\System\Streams> python adder.py < data.txt              sum file
1164

C:\...\PP4E\System\Streams> type data.txt | python adder.py        sum type output
1164

C:\...\PP4E\System\Streams> type writer2.py
for data in (123, 0, 999, 42):
    print('%03d' % data)

C:\...\PP4E\System\Streams> python writer2.py | python sorter.py   sort py output
000
042
123
999

C:\...\PP4E\System\Streams> writer2.py | sorter.py                 shorter form
...same output as prior command on Windows...

C:\...\PP4E\System\Streams> python writer2.py | python sorter.py | python adder.py
1164
```

The last command here connects three Python scripts by standard streams—the output of each prior script is fed to the input of the next via pipeline shell syntax.

Coding alternatives for adders and sorters

A few coding pointers here: if you look closely, you'll notice that *sorter.py* reads all of `stdin` at once with the `readlines` method, but *adder.py* reads one line at a time. If the input source is another program, some platforms run programs connected by pipes in *parallel*. On such systems, reading line by line works better if the data streams being passed are large, because readers don't have to wait until writers are completely finished to get busy processing data. Because `input` just reads `stdin`, the line-by-line scheme used by *adder.py* can always be coded with manual `sys.stdin` reads too:

```
C:\...\PP4E\System\Streams> type adder2.py
import sys
sum = 0
while True:
    line = sys.stdin.readline()
    if not line: break
    sum += int(line)
print(sum)
```

This version utilizes the fact that `int` allows the digits to be surrounded by whitespace (`readline` returns a line including its `\n`, but we don't have to use `[:-1]` or `rstrip()` to remove it for `int`). In fact, we can use Python's more recent file iterators to achieve the same effect—the `for` loop, for example, automatically grabs one line each time through when we iterate over a file object directly (more on file iterators in the next chapter):

```
C:\...\PP4E\System\Streams> type adder3.py
import sys
sum = 0
```

```
for line in sys.stdin: sum += int(line)
print(sum)
```

Changing `sorter` to read line by line this way may not be a big performance boost, though, because the list `sort` method requires that the list already be complete. As we'll see in Chapter 18, manually coded sort algorithms are generally prone to be much slower than the Python list sorting method.

Interestingly, these two scripts can also be coded in a much more compact fashion in Python 2.4 and later by using the new `sorted` built-in function, generator expressions, and file iterators. The following work the same way as the originals, with noticeably less source-file real estate:

```
C:\...\PP4E\System\Streams> type sorterSmall.py
import sys
for line in sorted(sys.stdin): print(line, end='')

C:\...\PP4E\System\Streams> type adderSmall.py
import sys
print(sum(int(line) for line in sys.stdin))
```

In its argument to `sum`, the latter of these employs a generator expression, which is much like a list comprehension, but results are returned one at a time, not in a physical list. The net effect is space optimization. For more details, see a core language resource, such as the book *Learning Python*.

Redirected Streams and User Interaction

Earlier in this section, we piped *teststreams.py* output into the standard `more` command-line program with a command like this:

```
C:\...\PP4E\System\Streams> python teststreams.py < input.txt | more
```

But since we already wrote our own "more" paging utility in Python in the preceding chapter, why not set it up to accept input from `stdin` too? For example, if we change the last three lines of the *more.py* file listed as Example 2-1 in the prior chapter...

```
if __name__ == '__main__':                      # when run, not when imported
    import sys
    if len(sys.argv) == 1:                      # page stdin if no cmd args
        more(sys.stdin.read())
    else:
        more(open(sys.argv[1]).read())
```

...it almost seems as if we should be able to redirect the standard output of *teststreams.py* into the standard input of *more.py*:

```
C:\...\PP4E\System\Streams> python teststreams.py < input.txt | python ..\more.py
Hello stream world
Enter a number>8 squared is 64
Enter a number>6 squared is 36
Enter a number>Bye
```

This technique generally works for Python scripts. Here, *teststreams.py* takes input from a file again. And, as in the last section, one Python program's output is piped to another's input—the *more.py* script in the parent (`..`) directory.

But there's a subtle problem lurking in the preceding `more.py` command. Really, chaining worked there only by sheer luck: if the first script's output is long enough that `more` has to ask the user if it should continue, the script will utterly fail (specifically, when `input` for user interaction triggers `EOFError`).

The problem is that the augmented *more.py* uses `stdin` for two disjointed purposes. It reads a reply from an interactive user on `stdin` by calling `input`, but now it *also* accepts the main input text on `stdin`. When the `stdin` stream is really redirected to an input file or pipe, we can't use it to input a reply from an interactive user; it contains only the text of the input source. Moreover, because `stdin` is redirected before the program even starts up, there is no way to know what it meant prior to being redirected in the command line.

If we intend to accept input on `stdin` *and* use the console for user interaction, we have to do a bit more: we would also need to use special interfaces to read user replies from a keyboard directly, instead of standard input. On Windows, Python's standard library `msvcrt` module provides such tools; on many Unix-like platforms, reading from device file */dev/tty* will usually suffice.

Since this is an arguably obscure use case, we'll delegate a complete solution to a suggested exercise. Example 3-8 shows a Windows-only modified version of the *more* script that pages the standard input stream if called with no arguments, but also makes use of lower-level and platform-specific tools to converse with a user at a keyboard if needed.

Example 3-8. PP4E\System\Streams\moreplus.py

```
"""
split and interactively page a string, file, or stream of
text to stdout; when run as a script, page stdin or file
whose name is passed on cmdline; if input is stdin, can't
use it for user reply--use platform-specific tools or GUI;
"""

import sys

def getreply():
    """
    read a reply key from an interactive user
    even if stdin redirected to a file or pipe
    """
    if sys.stdin.isatty():                      # if stdin is console
        return input('?')                       # read reply line from stdin
    else:
        if sys.platform[:3] == 'win':           # if stdin was redirected
            import msvcrt                       # can't use to ask a user
            msvcrt.putch(b'?')
```

```
            key = msvcrt.getche()               # use windows console tools
            msvcrt.putch(b'\n')                 # getch() does not echo key
            return key
        else:
            assert False, 'platform not supported'
            #linux?: open('/dev/tty').readline()[:-1]

def more(text, numlines=10):
    """
    page multiline string to stdout
    """
    lines = text.splitlines()
    while lines:
        chunk = lines[:numlines]
        lines = lines[numlines:]
        for line in chunk: print(line)
        if lines and getreply() not in [b'y', b'Y']: break

if __name__ == '__main__':                      # when run, not when imported
    if len(sys.argv) == 1:                      # if no command-line arguments
        more(sys.stdin.read())                  # page stdin, no inputs
    else:
        more(open(sys.argv[1]).read())          # else page filename argument
```

Most of the new code in this version shows up in its `getreply` function. The file's `isatty` method tells us whether `stdin` is connected to the console; if it is, we simply read replies on `stdin` as before. Of course, we have to add such extra logic only to scripts that intend to interact with console users *and* take input on `stdin`. In a GUI application, for example, we could instead pop up dialogs, bind keyboard-press events to run callbacks, and so on (we'll meet GUIs in Chapter 7).

Armed with the reusable `getreply` function, though, we can safely run our `moreplus` utility in a variety of ways. As before, we can import and call this module's function directly, passing in whatever string we wish to page:

```
>>> from moreplus import more
>>> more(open('adderSmall.py').read())
import sys
print(sum(int(line) for line in sys.stdin))
```

Also as before, when run with a command-line *argument*, this script interactively pages through the named file's text:

```
C:\...\PP4E\System\Streams> python moreplus.py adderSmall.py
import sys
print(sum(int(line) for line in sys.stdin))

C:\...\PP4E\System\Streams> python moreplus.py moreplus.py
"""
split and interactively page a string, file, or stream of
text to stdout; when run as a script, page stdin or file
whose name is passed on cmdline; if input is stdin, can't
use it for user reply--use platform-specific tools or GUI;
"""
```

```
import sys

def getreply():
?n
```

But now the script also correctly pages text redirected into `stdin` from either a *file* or a command *pipe*, even if that text is too long to fit in a single display chunk. On most shells, we send such input via redirection or pipe operators like these:

```
C:\...\PP4E\System\Streams> python moreplus.py < moreplus.py
"""
split and interactively page a string, file, or stream of
text to stdout; when run as a script, page stdin or file
whose name is passed on cmdline; if input is stdin, can't
use it for user reply--use platform-specific tools or GUI;
"""

import sys

def getreply():
?n

C:\...\PP4E\System\Streams> type moreplus.py | python moreplus.py
"""
split and interactively page a string, file, or stream of
text to stdout; when run as a script, page stdin or file
whose name is passed on cmdline; if input is stdin, can't
use it for user reply--use platform-specific tools or GUI;
"""

import sys

def getreply():
?n
```

Finally, piping one Python script's output into this script's input now works as expected, without botching user interaction (and not just because we got lucky):

```
......\System\Streams> python teststreams.py < input.txt | python moreplus.py
Hello stream world
Enter a number>8 squared is 64
Enter a number>6 squared is 36
Enter a number>Bye
```

Here, the standard *output* of one Python script is fed to the standard *input* of another Python script located in the same directory: *moreplus.py* reads the output of *teststreams.py*.

All of the redirections in such command lines work only because scripts don't care what standard input and output really are—interactive users, files, or pipes between programs. For example, when run as a script, *moreplus.py* simply reads stream `sys.stdin`; the command-line shell (e.g., DOS on Windows, csh on Linux) attaches such streams to the source implied by the command line before the script is started.

Scripts use the preopened `stdin` and `stdout` file objects to access those sources, regardless of their true nature.

And for readers keeping count, we have just run this single `more` pager script in four different ways: by importing and calling its function, by passing a filename command-line argument, by redirecting `stdin` to a file, and by piping a command's output to `stdin`. By supporting importable functions, command-line arguments, and standard streams, Python system tools code can be reused in a wide variety of modes.

Redirecting Streams to Python Objects

All of the previous standard stream redirections work for programs written in any language that hook into the standard streams and rely more on the shell's command-line processor than on Python itself. Command-line redirection syntax like `< filename` and `| program` is evaluated by the shell, not by Python. A more Pythonesque form of redirection can be done within scripts themselves by resetting `sys.stdin` and `sys.stdout` to file-like objects.

The main trick behind this mode is that anything that looks like a file in terms of methods will work as a standard stream in Python. The object's interface (sometimes called its protocol), and not the object's specific datatype, is all that matters. That is:

- Any object that provides file-like *read* methods can be assigned to `sys.stdin` to make input come from that object's read methods.
- Any object that defines file-like *write* methods can be assigned to `sys.stdout`; all standard output will be sent to that object's methods.

Because `print` and `input` simply call the `write` and `readline` methods of whatever objects `sys.stdout` and `sys.stdin` happen to reference, we can use this technique to both provide and intercept standard stream text with objects implemented as classes.

If you've already studied Python, you probably know that such plug-and-play compatibility is usually called *polymorphism*—it doesn't matter what an object is, and it doesn't matter what its interface does, as long as it provides the expected interface. This liberal approach to datatypes accounts for much of the conciseness and flexibility of Python code. Here, it provides a way for scripts to reset their own streams. Example 3-9 shows a utility module that demonstrates this concept.

Example 3-9. PP4E\System\Streams\redirect.py

```
"""
file-like objects that save standard output text in a string and provide
standard input text from a string; redirect runs a passed-in function
with its output and input streams reset to these file-like class objects;
"""

import sys                                      # get built-in modules

class Output:                                   # simulated output file
```