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.