10.2. Standard input, output, and error

UNIX users are already familiar with the concept of standard input, standard output, and standard error. This section is for the rest of you.

Standard output and standard error (commonly abbreviated `stdout` and `stderr`) are pipes that are built into every UNIX system. When you `print` something, it goes to the `stdout` pipe; when your program crashes and prints out debugging information (like a traceback in Python), it goes to the `stderr` pipe. Both of these pipes are ordinarily just connected to the terminal window where you are working, so when a program prints, you see the output, and when a program crashes, you see the debugging information. (If you're working on a system with a window-based Python IDE, `stdout` and `stderr` default to your "Interactive Window".)

Example 10.8. Introducing `stdout` and `stderr`

```
>>> for i in range(3):
...     print 'Dive in'              ❶
Dive in
Dive in
Dive in
>>> import sys
>>> for i in range(3):
...     sys.stdout.write('Dive in') ❷
Dive inDive inDive in
>>> for i in range(3):
...     sys.stderr.write('Dive in') ❸
Dive inDive inDive in
```

❶ As you saw in Example 6.9, "Simple Counters", you can use Python's built-in `range` function to build simple counter loops that repeat something a set number of times.

❷ `stdout` is a file-like object; calling its `write` function will print out whatever string you give it. In fact, this is what the `print` function really does; it adds a carriage return to the end of the string you're printing, and calls `sys.stdout.write`.

❸ In the simplest case, `stdout` and `stderr` send their output to the same place: the Python IDE (if you're in one), or the terminal (if you're running Python from the command line). Like `stdout`, `stderr` does not add carriage returns for you; if you want them, add them yourself.

`stdout` and `stderr` are both file-like objects, like the ones you discussed in Section 10.1, "Abstracting input sources", but they are both write-only. They have no `read` method, only `write`. Still, they are file-like objects, and you can

assign any other file- or file-like object to them to redirect their output.

Example 10.9. Redirecting output

```
[you@localhost kgp]$ python stdout.py
Dive in
[you@localhost kgp]$ cat out.log
This message will be logged instead of displayed
```

(On Windows, you can use `type` instead of `cat` to display the contents of a file.)

If you have not already done so, you can download this and other examples used in this book.

```
#stdout.py
import sys

print 'Dive in'                                          ❶
saveout = sys.stdout                                    ❷
fsock = open('out.log', 'w')                            ❸
sys.stdout = fsock                                      ❹
print 'This message will be logged instead of displayed' ❺
sys.stdout = saveout                                    ❻
fsock.close()                                           ❼
```

❶ This will print to the IDE "Interactive Window" (or the terminal, if running the script from the command line).

❷ Always save `stdout` before redirecting it, so you can set it back to normal later.

❸ Open a file for writing. If the file doesn't exist, it will be created. If the file does exist, it will be overwritten.

❹ Redirect all further output to the new file you just opened.

❺ This will be "printed" to the log file only; it will not be visible in the IDE window or on the screen.

❻ Set `stdout` back to the way it was before you mucked with it.

❼ Close the log file.

Redirecting `stderr` works exactly the same way, using `sys.stderr` instead of `sys.stdout`.

Example 10.10. Redirecting error information

```
[you@localhost kgp]$ python stderr.py
[you@localhost kgp]$ cat error.log
Traceback (most recent line last):
  File "stderr.py", line 5, in ?
    raise Exception, 'this error will be logged'
Exception: this error will be logged
```

If you have not already done so, you can download this and other examples used in this book.

```
#stderr.py
import sys
```

```
fsock = open('error.log', 'w')              ❶
sys.stderr = fsock                          ❷
raise Exception, 'this error will be logged' ❸ ❹
```

❶ Open the log file where you want to store debugging information.

❷ Redirect standard error by assigning the file object of the newly-opened log file to `stderr`.

❸ Raise an exception. Note from the screen output that this does *not* print anything on screen. All the normal traceback information has been written to `error.log`.

❹ Also note that you're not explicitly closing your log file, nor are you setting `stderr` back to its original value. This is fine, since once the program crashes (because of the exception), Python will clean up and close the file for us, and it doesn't make any difference that `stderr` is never restored, since, as I mentioned, the program crashes and Python ends. Restoring the original is more important for `stdout`, if you expect to go do other stuff within the same script afterwards.

Since it is so common to write error messages to standard error, there is a shorthand syntax that can be used instead of going through the hassle of redirecting it outright.

Example 10.11. Printing to `stderr`

```
>>> print 'entering function'
entering function
>>> import sys
>>> print >> sys.stderr, 'entering function' ❶
entering function
```

❶ This shorthand syntax of the `print` statement can be used to write to any open file, or file-like object. In this case, you can redirect a single `print` statement to `stderr` without affecting subsequent `print` statements.

Standard input, on the other hand, is a read-only file object, and it represents the data flowing into the program from some previous program. This will likely not make much sense to classic Mac OS users, or even Windows users unless you were ever fluent on the MS-DOS command line. The way it works is that you can construct a chain of commands in a single line, so that one program's output becomes the input for the next program in the chain. The first program simply outputs to standard output (without doing any special redirecting itself, just doing normal `print` statements or whatever), and the next program reads from standard input, and the operating system takes care of connecting one program's output to the next program's input.

Example 10.12. Chaining commands

```
[you@localhost kgp]$ python kgp.py -g binary.xml        ❶
01100111
[you@localhost kgp]$ cat binary.xml                     ❷
<?xml version="1.0"?>
<!DOCTYPE grammar PUBLIC "-//diveintopython.org//DTD Kant Generator Pro v1.0//EN" "kgp.dtd">
<grammar>
<ref id="bit">
  <p>0</p>
  <p>1</p>
</ref>
<ref id="byte">
```

```
  <p><xref id="bit"/><xref id="bit"/><xref id="bit"/><xref id="bit"/>\
<xref id="bit"/><xref id="bit"/><xref id="bit"/><xref id="bit"/></p>
</ref>
</grammar>
[you@localhost kgp]$ cat binary.xml | python kgp.py -g - ❸ ❹
10110001
```

❶ As you saw in Section 9.1, "Diving in", this will print a string of eight random bits, `0` or `1`.

❷ This simply prints out the entire contents of `binary.xml`. (Windows users should use `type` instead of `cat`.)

❸ This prints the contents of `binary.xml`, but the "`|`" character, called the "pipe" character, means that the contents will not be printed to the screen. Instead, they will become the standard input of the next command, which in this case calls your Python script.

❹ Instead of specifying a module (like `binary.xml`), you specify "`-`", which causes your script to load the grammar from standard input instead of from a specific file on disk. (More on how this happens in the next example.) So the effect is the same as the first syntax, where you specified the grammar filename directly, but think of the expansion possibilities here. Instead of simply doing `cat binary.xml`, you could run a script that dynamically generates the grammar, then you can pipe it into your script. It could come from anywhere: a database, or some grammar-generating meta-script, or whatever. The point is that you don't need to change your `kgp.py` script at all to incorporate any of this functionality. All you need to do is be able to take grammar files from standard input, and you can separate all the other logic into another program.

So how does the script "know" to read from standard input when the grammar file is "`-`"? It's not magic; it's just code.

Example 10.13. Reading from standard input in `kgp.py`

```
def openAnything(source):
    if source == "-":    ❶
        import sys
        return sys.stdin

    # try to open with urllib (if source is http, ftp, or file URL)
    import urllib
    try:

[... snip ...]
```

❶ This is the `openAnything` function from `toolbox.py`, which you previously examined in Section 10.1, "Abstracting input sources". All you've done is add three lines of code at the beginning of the function to check if the source is "`-`"; if so, you return `sys.stdin`. Really, that's it! Remember, `stdin` is a file-like object with a `read` method, so the rest of the code (in `kgp.py`, where you call `openAnything`) doesn't change a bit.

Report issues