ГЛАВА 3

Контекст выполнения сценариев

"Я хотел бы иметь аргумент"

Сценарии Python не работают в вакууме (несмотря на то, что вы, возможно, слышали иное). В зависимости от платформ и процедур запуска программы на Python могут иметь всевозможный объемлющий контекст — информацию, автоматически передаваемую программе операционной системой (ОС) при ее запуске. Например, сценарии имеют доступ к описанным ниже видам входных данных и интерфейсов системного уровня.

- *Текущий рабочий каталог.* Метод `os.getcwd` предоставляет доступ к каталогу, из которого был запущен сценарий, и данное значение неявно используется многими файловыми инструментами.
- *Аргументы командной строки.* Объект `sys.argv` предоставляет доступ к словам, набранным в командной строке, которые применяются при запуске программы и служат входными данными сценария.
- *Переменные среды.* Объект `os.environ` предлагает интерфейс к именам, которым присвоены значения в объемлющей оболочке (или в дочерней программе) и которые передаются сценарию.
- *Стандартные потоки ввода-вывода.* Объекты `sys.stdin`, `stdout` и `stderr` экспортируют потоки ввода-вывода, которые лежат в основе инструментов командной строки оболочки и могут быть задействованы в сценариях через параметры функции `print`, вызов `os.popen`, модуль `subprocess`, представленный в главе 2, класс `io.StringIO` и многие другие средства.

Перечисленные инструменты могут служить входными данными для сценариев, параметров конфигурации и т.д. В настоящей главе мы рассмотрим все четыре контекстных инструмента — как их интерфейсы Python, так и их типичные роли.

Текущий рабочий каталог

Понятие текущего рабочего каталога (current working directory — CWD) оказывается ключевым при выполнении некоторых сценариев: оно представляет собой неявное место, где предполагается размещение файлов, обрабатываемых сценарием, если в их именах не указаны абсолютные пути к каталогам. Как мы видели ранее, `os.getcwd` позволяет сценарию явно получать имя CWD, а `os.chdir` дает возможность сценарию перемещаться в новый CWD.

Однако имейте в виду, что имена файлов без полных путей отображаются на CWD и не имеют ничего общего с имеющейся настройкой `PYTHONPATH`. Формально сценарий всегда запускается из CWD, а не из каталога, содержащего файл сценария. И наоборот, при импортировании сначала производится поиск в каталоге, содержащем сценарий, а не в CWD (если только сценарий не находится в CWD). Поскольку это тонкое различие может сбить с толку новичков, давайте рассмотрим его более подробно.

Текущий рабочий каталог, файлы и пути импортирования

Если вы запустите сценарий Python, вводя командную строку оболочки вроде `python dir1\dir2\file.py`, то в качестве CWD будет выступать каталог, в котором вы находились, когда вводили команду, а не `dir1\dir2`. С другой стороны, интерпретатор Python автоматически добавляет идентификатор домашнего каталога сценария в начало пути поиска импортируемых модулей, так что сценарий `file.py` всегда может импортировать другие файлы в `dir1\dir2` независимо от того, откуда он запускается. В целях иллюстрации давайте напишем простой сценарий, отображающий как его CWD, так и его путь поиска импортируемых модулей:

```
C:\...\PP4E\System> type whereami.py
import os, sys
print('my os.getcwd =>', os.getcwd())   # показать текущий рабочий каталог
print('my sys.path =>', sys.path[:6])   # показать первые 6 путей импортирования
input()                                 # ожидать нажатия клавиши
```

Теперь при запуске сценария `whereami.py` из каталога, в котором он находится, ожидаемым образом устанавливается каталог CWD, который добавляется в начало пути поиска импортируемых модулей. Путь поиска импортируемых модулей `sys.path` вы уже встречали ранее; его первая запись также может быть пустой строкой для обозначения CWD во время работы в интерактивном режиме, а большая часть CWD при отображении усекается до `...`:

```
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', ...и т.д...]
```

Но в случае запуска сценария `whereami.py` из других мест каталог CWD тоже перемещается (это каталог, в котором вводятся команды), и интерпретатор Python добавляет каталог в начало пути поиска импортируемых модулей, что позволяет сценарию по-прежнему видеть файлы в собственном домашнем каталоге. Например, при запуске на один уровень выше (`..`) имя `System`,

добавленное в начало sys.path, будет первым каталогом, где Python ищет импортируемые модули в whereami.py; оно нацеливает обратно на каталог, содержащий запущенный сценарий. Тем не менее, имена файлов без полных путей будут отображаться на каталог CWD (C:\PP4thEd\Examples\PP4E), а не на вложенный в него подкаталог System:

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

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', ...]
```

Совокупный эффект заключается в том, что имена файлов без путей к каталогам в сценарии будут сопоставляться с местом, где была введена *команда* (os.getcwd), но операторы импортирования по-прежнему имеют доступ к каталогу выполняющегося *сценария* (через начало в sys.path). Наконец, когда файл запускается щелчком на его значке, CWD — это просто каталог, содержащий выбранный файл. Например, при двойном щелчке на файле whereami.py внутри проводника Windows в новом окне консоли DOS появляется следующий вывод:

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

В данном случае каталог CWD, используемый для имен файлов, и первый каталог поиска импортируемых модулей представляют собой каталог, содержащий файл сценария. Все это обычно работает именно так, как вы ожидаете, но следует избегать двух ловушек.

- Имена файлов могут включать полные пути к каталогам, если нет уверенности в том, откуда будут запускаться сценарии.
- Сценарии командной строки не всегда могут полагаться на CWD для обеспечения видимости при импортировании файлов, которые не находятся в собственных каталогах; взамен для доступа к модулям в других каталогах необходимо использовать настройки PYTHONPATH и пути импортирования пакетов.

Скажем, приведенные в книге сценарии вне зависимости от того, как они запускаются, всегда могут импортировать другие файлы в собственные домашние каталоги без указания пути к импортируемым пакетам (import файл здесь), но должны проходить через корень пакета PP4E, чтобы найти файлы где-то в другом месте дерева примеров (from PP4E.dir1.dir2 import файл там), даже если они запускаются из каталога, содержащего нужный внешний модуль. Как обычно для модулей, имя каталога PP4E\dir1\dir2 тоже может быть добавлено в PYTHONPATH, делая файлы видимыми везде без пути к импортируемым пакетам (хотя добавление дополнительных каталогов в PYTHONPATH увеличивает вероятность конфликтов имен). Однако в любом случае импортирование всегда осуществляется из домашнего каталога сценария либо из других каталогов в пути поиска Python, а не из CWD.

Текущий рабочий каталог и командные строки

Описанное выше различие между CWD и путями поиска импортируемых модулей объясняет, почему многие сценарии в книге, разработанные для оперирования в текущем рабочем каталоге (а не в том, имя которого передается), запускаются с помощью командных строк следующего вида:

```
C:\temp> python C:\...\PP4E\Tools\cleanpyc.py обработать текущий рабочий каталог
```

В показанном примере сам файл сценария Python находится в каталоге `C:\...\PP4E\Tools`, но из-за того, что сценарий запускается из `C:\temp`, он обрабатывает файлы, расположенные в `C:\temp` (т.е. в CWD, а не в своем домашнем каталоге). Для обработки с помощью такого сценария файлов, расположенных в другом месте, понадобится просто перейти в соответствующий каталог, чтобы изменить CWD:

```
C:\temp> cd C:\PP4thEd\Examples
C:\PP4thEd\Examples> python C:\...\PP4E\Tools\cleanpyc.py      обработать CWD
```

Так как CWD всегда подразумевается, команда `cd` сообщает сценарию, какой каталог обрабатывать, в не менее определенных терминах, чем явная передача имени каталога сценарию (примечание относительно переносимости: для предотвращения развертывания шаблона `*.py` в некоторых оболочках Unix могут потребоваться кавычки вокруг `*.py` здесь и в других примерах командной строки):

```
C:\...\PP4E\Tools> python find.py *.py C:\temp обработать каталог с указанным именем
```

В приведенной выше командной строке CWD — это каталог, содержащий запускаемый сценарий (обратите внимание, что имя файла сценария не имеет префикса пути к каталогу); но поскольку сценарий обрабатывает каталог, указанный явно в командной строке (`C:\temp`), то CWD не имеет значения. Наконец, если мы хотим запустить такой сценарий, находящийся в другом каталоге, для обработки файлов, расположенных в еще одном каталоге, то можем просто указать пути к обоим каталогам:

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

Здесь сценарий получает видимость для импорта в файлы из своего домашнего каталога `PP4E\Tools` и обрабатывает файлы в каталоге, указанном в командной строке, но CWD представляет собой нечто совершенно другое (`C:\temp`). Конечно, последняя форма предусматривает больший объем набора, но обратите внимание в книге на различные CWD и явные командные строки путей к сценариям.

Аргументы командной строки

Модуль `sys` также позволяет получить доступ к словам, набранным в команде, которая применяется для запуска сценария Python. Такие слова обычно называются аргументами командной строки и находятся во встроенном списке строк `sys.argv`. Программисты на языке C могут заметить его сходство с массивом `argv` (массивом строк C). В интерактивном режиме смотреть особо не на что, т.к. для запуска Python в этом режиме аргументы командной строки не передаются:

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

Чтобы действительно взглянуть на аргументы, необходимо запустить сценарий из командной строки оболочки. В примере 3.1 показан простейший сценарий, который выводит список `argv` для инспектирования.

Пример 3.1. PP4E\System\testargv.py

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

В результате выполнения сценария `testargv.py` выводится список аргументов командной строки; обратите внимание, что первым элементом всегда является имя самого исполняемого файла сценария Python независимо от способа его запуска (см. врезку "Исполняемые сценарии в Unix" далее в главе).

```
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']
```

Последняя команда иллюстрирует общее соглашение. Как и аргументы функций, параметры командной строки иногда передаются по позиции, а иногда по имени с использованием пары `-имя значение`. Скажем, пара `-i data.txt` указывает на то, что параметр `-i` имеет значение `data.txt` (например, имя входного файла). Передавать можно любые слова, но обычно программы придают им определенную структурированность.

Аргументы командной строки играют в программах ту же роль, что и аргументы функций в функциях: они являются способом передачи информации в программу, которая может меняться в зависимости от запуска программы. Поскольку они не должны быть жестко закодированы, появляется возможность применения сценариев в более обобщенном плане. Скажем, сценарий обработки файлов может использовать аргумент командной строки в качестве имени файла, который он должен обработать; примером может служить сценарий `more.py` в главе 2 (пример 2.1). Другие сценарии могут принимать флаги режима обработки, Интернет-адреса и т.п.

Передача аргументов командной строки

Тем не менее, как только вы начнете регулярно использовать аргументы командной строки, вам, скорее всего, покажется неудобным продолжать писать код, который просматривает список в поиске слов. Как правило, программы преобразуют список аргументов при запуске в более удобные для обработки структуры. Вот один из способов сделать это: сценарий в примере 3.2 просматривает список `argv` в поиске пар слов `-имя_параметра значение_параметра` и помещает их в словарь по именам параметров для облегчения поиска.

Пример 3.2. PP4E\System\testargv2.py

```
"помещение параметров командной строки в словарь"
def getopts(argv):
    opts = {}
    while argv:
        if argv[0][0] == '-':           # найти пары "-имя значение"
            opts[argv[0]] = argv[1]     # ключом словаря является аргумент "-имя"
            argv = argv[2:]
        else:
            argv = argv[1:]
    return opts
if __name__== '__main__':
    from sys import argv                # пример клиентского кода
    myargs = getopts(argv)
    if '-i' in myargs:
        print(myargs['-i'])

    print(myargs)
```

Можете импортировать и использовать такую функцию во всех инструментах командной строки. При запуске сам по себе файл `testargv2.py` просто выводит форматированный словарь аргументов:

```
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'}
```

Естественно, мы могли бы предусмотреть более изощренную реализацию шаблонов аргументов, проверку ошибок и т.п. В случае сложных командных строк для разбора аргументов мы также можем применять инструменты обработки командной строки из стандартной библиотеки Python:

- модуль `getopt`, созданный по образцу одноименной утилиты Unix/C;
- модуль `optparse`, который является новейшей альтернативой и считается более мощным.

Оба модуля документированы в руководстве по библиотеке Python, где также приведены примеры их использования, которые здесь не повторяются ради экономии места. В общем, чем более настраиваемые ваши сценарии, тем больше вам придется вкладывать в реализацию логики обработки командной строки.

Исполняемые сценарии в Unix

Пользователи Unix и Linux могут сделать текстовые файлы с исходным кодом Python непосредственно исполняемыми, добавив в их начало специальную строку с путем к интерпретатору Python и предоставив файлу разрешение исполняемого файла. Например, поместите следующий код в текстовый файл по имени `myscript`:

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

Первая строка обычно воспринимается интерпретатором Python как комментарий (она начинается с #); но в случае запуска этого файла ОС отправляет его строки интерпретатору, указанному после #!. Как только файл myscript будет сделан непосредственно исполняемым с помощью команды оболочки chmod +x myscript, его можно будет запускать напрямую, не вводя python в команде, как если бы он был двоичной исполняемой программой:

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

При таком запуске sys.argv по-прежнему будет иметь имя сценария в качестве первого слова в списке: ["myscript", "a", "b", "c"] в точности, как если бы сценарий был запущен с помощью более явной и переносимой формы команды python myscript a b c. Превращение сценариев в непосредственно исполняемые на самом деле является трюком Unix, а не функцией Python, но стоит отметить, что его можно сделать чуть менее зависимым от машины, указав в начале сценария Unix-команду env вместо жестко закодированного пути к интерпретатору Python:

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

В таком случае для поиска интерпретатора Python будет использоваться переменная среды (PATH на большинстве платформ). Если один и тот же сценарий необходимо запускать на многих машинах, то понадобится лишь изменить настройки среды на каждой машине (редактировать код сценария Python не нужно). Разумеется, файлы Python всегда можно запускать посредством более явной командной строки:

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

Здесь предполагается, что интерпретатор Python находится в настройках пути поиска вашей системы (иначе придется ввести полный путь), но представленная команда работает на любой платформе Python с командной строкой. По причине лучшей переносимости такое соглашение обычно применяется в примерах, рассматриваемых в книге, но для получения более подробной информации по любой из упомянутых здесь тем обращайтесь к справочным страницам Unix. Тем не менее, специальные строки #! будут встречаться во многих примерах на тот случай, если читатели захотят запускать их в виде исполняемых файлов в Unix или Linux; на других платформах они просто игнорируются как комментарии Python.

Обратите внимание, что в последних версиях Windows обычно можно вводить имя файла сценария напрямую (без слова python) для его запуска, и добавлять строку #! в начало файла не требуется. На этой платформе интерпретатор Python использует реестр Windows, объявляя себя программой, которая открывает файлы с расширениями Python (.py и другие). По той же причине имеется возможность запускать файлы в Windows, щелкая на их значках.

Переменные среды оболочки

Переменные оболочки, иногда называемые переменными среды, доступны для сценариев Python как `os.environ` — объект, подобный словарю Python, с одной записью для каждой переменной в оболочке. Переменные оболочки существуют вне системы Python; они часто устанавливаются в системной подсказке или внутри файлов запуска либо в графическом пользовательском интерфейсе панели управления и обычно служат общесистемными конфигурационными входными данными для программ.

Фактически к настоящему моменту вы уже должны быть знакомы с основным примером: настройка пути поиска модулей `PYTHONPATH` является переменной оболочки, используемой интерпретатором Python для импортирования модулей. После установки `PYTHONPATH` в среде ОС ее значение будет доступно каждый раз при запуске программ Python. Переменные оболочки также могут быть установлены программами, чтобы они служили входными данными для других программ в приложении; поскольку их значения обычно наследуются порожденными программами, их можно применять как простую форму взаимодействия между процессами.

Получение переменных оболочки

В Python окружающая среда оболочки становится простым предустановленным объектом, доступ к которому не требует использования специального синтаксиса. Индексация `os.environ` по строке с именем желаемой переменной оболочки (например, `os.environ['USER']`) является эквивалентом добавления знака доллара перед именем переменной в большинстве оболочек Unix (скажем, `$USER`), применения символов процента с обеих сторон имени в DOS (`%USER%`) и вызова `getenv("USER")` в программе на C. Давайте запустим интерактивный сеанс и поэкспериментируем (далее запускается интерпретатор Python 3.11 на компьютере с Windows 10):

```
   >>> import os
   >>> os.environ.keys()
   KeysView(environ({'ALLUSERSPROFILE': 'C:\\ProgramData', 'APPDATA':
'C:\\Users\\ASUS-Users\\AppData\\Roaming',
   ...остальная информация не показана...

   >>> list(os.environ.keys())
   ['ALLUSERSPROFILE', 'APPDATA', 'COMMONPROGRAMFILES', 'COMMONPROGRAMFILES(X86)',
'COMMONPROGRAMW6432', 'COMPUTERNAME',
   ...остальная информация не показана...
   'NUMBER_OF_PROCESSORS', 'ONEDRIVE', 'OS', 'PATH', 'PATHEXT',
'PROCESSOR_ARCHITECTURE', 'PROCESSOR_IDENTIFIER', 'PROCESSOR_LEVEL',
   ...остальная информация не показана...

   >>> os.environ['TEMP']
   'C:\\Users\\ASUS-U~1\\AppData\\Local\\Temp'
```

Метод `keys` возвращает итерируемый объект с присвоенными переменными, а индексация извлекает значение переменной среды `TEMP` в Windows. Такой прием работает и в среде Linux, но при запуске Python обычно предварительно устанавливаются другие переменные. Поскольку нам уже известна переменная среды `PYTHONPATH`, давайте проверим ее содержимое:

```
>>> os.environ['PYTHONPATH']
'D:\\PP4thEd\\Examples;C:\\Users\\ASUS-Users\\temp'
>>> for srcdir in os.environ['PYTHONPATH'].split(os.pathsep):
...     print(srcdir)
...
D:\\PP4thEd\\Examples C:\\Users\\ASUS-Users\\temp
>>> import sys
>>> sys.path[:3]
['', 'D:\\PP4thEd\\Examples', 'C:\\Users\\ASUS-Users\\temp']
```

Переменная среды PYTHONPATH представляет собой строку с путями к каталогам, разделенными любым символом, который используется для разделения элементов в таких путях на имеющейся платформе (например, `;` в DOS/Windows или `:` в Unix и Linux). Чтобы разделить ее на компоненты, мы передаем строковому методу split разделитель os.pathsep — переносимую настройку, которая дает корректный разделитель для базовой машины. Как обычно, sys.path является фактическим путем поиска во время выполнения и отражает результат объединения текущего каталога и настройки PYTHONPATH.

Изменение переменных оболочки

Аналогично нормальным словарям объект os.environ поддерживает как индексацию, так и *присваивание* ключей. Присваивание в словаре изменяет значение ключа:

```
>>> os.environ['TEMP']
'C:\\Users\\ASUS-U~1\\AppData\\Local\\Temp'
>>> os.environ['TEMP'] = r'c:\temp'
>>> os.environ['TEMP']
'c:\\temp'
```

Но здесь происходит кое-что еще. Во всех последних выпусках Python значения, присвоенные подобным образом ключам os.environ, автоматически *экспортируются* в другие части приложения. То есть присваивание значений ключам изменяет как объект os.environ в программе на Python, так и связанную с ним переменную в окружающей *оболочке* процесса запущенной программы. Новое значение становится видимым для программы на Python, всех связанных модулей C и любых программ, порожденных процессом Python.

Внутренне присваивание значений ключам в os.environ приводит к вызову функции os.putenv, которая изменяет переменную оболочки за пределами интерпретатора Python. Чтобы продемонстрировать, как это работает, нам понадобится пара сценариев, которые устанавливают и извлекают переменные оболочки; первый сценарий показан в примере 3.3.

Пример 3.3. PP4E\System\Environment\setenv.py

```
import os
print('setenv...', end=' ')
print(os.environ['USER'])       # показать текущее значение переменной оболочки

os.environ['USER'] = 'Brian'    # внутренне выполняет функцию os.putenv
os.system('python echoenv.py')
```

```
os.environ['USER'] = 'Arthur'     # изменения передаются порожденным программам
os.system('python echoenv.py')    # и связанным библиотечным модулям C

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

Приведенный сценарий `setenv.py` просто изменяет переменную оболочки `USER` и запускает еще один сценарий, который выводит значение этой переменной и представлен в примере 3.4.

Пример 3.4. PP4E\System\Environment\echoenv.py

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

Независимо от того, как запускается сценарий `echoenv.py`, он отображает значение переменной `USER` в окружающей оболочке; при запуске из командной строки выводится значение, установленное для данной переменной в самой оболочке:

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

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

Однако при запуске из другого сценария вроде `setenv.py` с помощью инструментов `os.system` и `os.popen`, которые упоминались ранее, `echoenv.py` получает значение `USER`, присвоенное в его родительской программе:

```
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
```

В Linux прием работает аналогично. Выражаясь в общих чертах, порожденная программа всегда *наследует* настройки среды от своих родителей. *Порожденные* программы — это программы, запущенные с помощью инструментов Python, таких как `os.spawnv`, комбинации `os.fork/exec` на Unix-подобных платформах, а также `os.popen`, `os.system` и модуля `subprocess` на различных платформах. Все программы, запускаемые подобным образом, получают настройки переменных среды, существующие в родительской программе на момент запуска *.

* Так принято по умолчанию. Есть инструменты запуска программ, которые позволяют сценариям передавать дочерним программам параметры среды, отличающиеся от собственных параметров. Например, вызов `os.spawnve` подобен `os.spawnv`, но он принимает аргумент типа словаря, представляющий среду оболочки, которая передается запускаемой программе. Некоторые варианты `os.exec*` (с буквой `e` в конце имени) аналогичным образом принимают

явное определение среды; более подробную информацию ищите в главе 5, где описаны форматы вызовов os.exec*.

С более широкой точки зрения установка таких переменных оболочки перед запуском новой программы является одним из способов передачи информации в новую программу. Например, сценарий конфигурирования Python может адаптировать переменную PYTHONPATH с целью включения специальных каталогов непосредственно перед запуском другого сценария Python. Запущенный сценарий будет иметь специальный путь поиска в своем sys.path, потому что переменные оболочки передаются дочерним элементам (сценарий запуска подобного рода встретится в конце главы 6).

Особенности переменных оболочки: родители, putenv и getenv

В предыдущем примере обратите внимание на последнюю команду: переменная USER возвращается к исходному значению после выхода из программы Python верхнего уровня. Результаты присваивания ключей в os.environ передаются за пределы интерпретатора *вниз* по цепочке порожденных программ, но никогда не возвращаются *вверх* к процессам родительской программы (включая системную оболочку). Это справедливо и для программ на C, использующих библиотечный вызов putenv, и само по себе не является ограничением языка Python.

Вряд ли это окажется проблемой, если сценарий Python запускается в начале приложения. Но имейте в виду, что настройки оболочки, сделанные внутри программы, обычно остаются актуальными только во время выполнения этой программы и ее порожденных потомков. Если вам нужно экспортировать настройку переменной оболочки, чтобы она сохранялась после выхода из Python, тогда можете найти соответствующие расширения, специфичные для платформы (см. http://www.python.org).

Есть еще одна тонкость: в современной реализации изменения в os.environ приводят к автоматическому вызову os.putenv, который запускает вызов putenv из библиотеки C (если она доступна на вашей платформе) для экспорта настройки за пределы Python в любой связанный код C. Тем не менее, хотя изменения os.environ инициируют обращение к os.putenv, прямые вызовы os.putenv не обновляют os.environ, чтобы отразить изменение. Из-за этого интерфейс сопоставления с os.environ обычно предпочтительнее os.putenv.

Также обратите внимание, что настройки среды загружаются в os.environ при запуске, а не при каждом извлечении; следовательно, изменения, внесенные связанным кодом C после запуска, могут не отражаться в os.environ. В настоящее время в Python имеется более целенаправленный вызов os.getenv, но на множестве платформ (или на всех в версии 3.X) он просто транслируется в вызов os.environ, а не в вызов getenv из библиотеки C. Для большинства приложений это нормально, особенно если они представляют собой чистый код Python. На платформах без putenv вызов os.environ может передаваться в качестве параметра инструментам запуска программы для установки среды порожденной программы.

Стандартные потоки ввода-выода

В модуле sys также находятся стандартные потоки ввода, вывода и ошибок для программ на Python, которые являются еще одним распространенным способом взаимодействия программ:

```
>>> import sys
>>> for f in (sys.stdin, sys.stdout, sys.stderr): print(f)
...
<_io.TextIOWrapper name='<stdin>' mode='r' encoding='utf-8'>
<_io.TextIOWrapper name='<stdout>' mode='w' encoding='utf-8'>
<_io.TextIOWrapper name='<stderr>' mode='w' encoding='utf-8'>
```

Стандартные потоки представляют собой предварительно открытые файловые объекты Python, которые автоматически подключаются к стандартным потокам ввода-вывода программы при запуске интерпретатора Python. По умолчанию все они привязаны к окну консоли, в котором был запущен интерпретатор Python (или программа на Python). Так как встроенные функции print и input на самом деле являются просто удобными интерфейсами для стандартных потоков вывода и ввода, они аналогичны прямому использованию stdout и stdin в модуле sys:

```
>>> 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'
```

Стандартные потоки ввода-вывода в Windows

Пользователи Windows для запуска программы на Python могут дважды щелкнуть на имени ее файла с расширением .py в проводнике (или запустить ее с помощью os.system). В результате откроется окно консоли DOS, которое будет служить стандартным потоком ввода-вывода программы. Если программа создает собственные окна, тогда вы можете избежать всплывающего окна консоли, назначив файлу исходного кода программы расширение .pyw вместо .py. Расширение .pyw обозначает исходный файл .py, не требующий открытия всплывающего окна DOS в Windows (для запуска используются настройки реестра Windows). Файл .pyw также можно импортировать обычным образом.

Вдобавок имейте в виду, что поскольку при запуске сценария двойным щелчком на его имени вывод отправляется во всплывающее окно DOS, выполнение сценария, который просто выводит текст и завершает работу, будет выглядеть несколько странно: всплывающее окно консоли DOS открывается, в него выводится текст, после чего всплывающее окно сразу исчезает (не самое дружественное поведение!). Для сохранения всплывающего окна DOS, чтобы можно было просмотреть вывод, необходимо просто добавить в конец сценария вызов input(), который приостановит его завершение до нажатия клавиши <Enter>.

Перенаправление потоков в файлы и программы

Формально стандартный текст вывода (и вывод с помощью `print`) отображается в окне консоли, из которого была запущена программа, стандартный текст ввода (и ввод посредством `input`) поступает с клавиатуры, а стандартный текст ошибок используется для вывода сообщений об ошибках в окно консоли. Во всяком случае, так принято по умолчанию. Кроме того, эти потоки можно *перенаправлять* как в файлы, так и в другие программы в системной оболочке, а также на произвольные объекты внутри сценария Python. В большинстве систем такие перенаправления упрощают повторное использование и объединение утилит командной строки общего назначения.

Перенаправление удобно применять для готовых (предварительно закодированных) тестовых входных данных: один и тот же тестовый сценарий можно использовать с любым набором входных данных, просто перенаправляя стандартный поток ввода в другой файл каждый раз при запуске сценария. Точно так же перенаправление стандартного потока вывода позволяет сохранять и позже анализировать вывод программы; например, системы тестирования могут сравнивать сохраненный стандартный вывод сценария с файлом ожидаемого вывода с целью обнаружения отказов.

Несмотря на всю мощь этой парадигмы, перенаправление оказывается простым в применении. Рассмотрим программу с простым циклом типа "прочитать-вычислить-вывести" в примере 3.5.

Пример 3.5. PP4E\System\Streams\teststreams.py

```
"чтение чисел до конца файла и отображение их квадратов"

def interact():
    print('Hello stream world')                # print выводит в sys.stdout
    while True:
        try:
            reply = input('Enter a number>')   # Введите число>
                                               # input читает sys.stdin
        except EOFError:
            break      # сгенерировать исключение, если встретился конец файла
        else:                                  # входные данные в виде строки
            num = int(reply)
            print("%d squared is %d" % (num, num ** 2))
                 # %d после возведения в квадрат равно %d
    print('Bye')                               # Работа завершена

if __name__=='__main__':
    interact()                                 # при запуске, но не импортировании
```

Как обычно, функция `interact` автоматически выполняется при запуске этого файла, а не при его импортировании. По умолчанию при запуске файла `teststreams.py` из командной строки данный стандартный поток появляется там, где была введена команда Python. Сценарий просто читает числа, пока не будет достигнут конец файла в стандартном потоке ввода (в Windows конец

файла обычно представляет собой комбинацию клавиш <Ctrl+Z>, а в Unix — <Ctrl+D>[1]):

```
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
```

Как в среде Windows, так и на Unix-подобных платформах стандартный поток ввода можно перенаправлять, чтобы он поступал из файла посредством синтаксиса оболочки < имя_файла. Ниже представлен сеанс в окне консоли DOS, который заставляет сценарий читать ввод из текстового файла `input.txt`. Того же самого можно добиться в Linux, но DOS-команду `type` понадобится заменить Unix-командой `cat`:

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

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

Файл `input.txt` автоматизирует ввод, который обычно набирается в интерактивном режиме — сценарий выполняет чтение из файла `input.txt`, а не с клавиатуры. Стандартный вывод можно аналогичным образом перенаправлять в файл с помощью синтаксиса оболочки > имя_файла. На самом деле перенаправление ввода и вывода можно применять в одной команде:

```
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
```

На этот раз входные и выходные данные сценария Python помещаются в текстовые файлы, а не в сеанс интерактивной консоли.

[1] Обратите внимание, что функция `input` генерирует исключение, чтобы сигнализировать о конце файла, но методы чтения файла для этого условия просто возвращают пустую строку. Поскольку `input` также удаляет символ конца строки, результирующая пустая строка означает пустую строчку в файле, поэтому для указания условия конца файла необходимо сгенерировать исключение. Методы чтения файлов сохраняют символ конца строки и обозначают пустую строчку как `"\n"` вместо `""`. Это один из способов, которым чтение `sys.stdin` напрямую отличается от `input`. Функция `input` также принимает строку приглашения, которая автоматически отображается перед принятием вводимых данных.

Соединение программ в цепочки с помощью каналов

В среде Windows и на Unix-подобных платформах стандартный вывод одной программы можно отправлять стандартному вводу другой, указывая между двумя командами символ оболочки |. В таком случае оболочка создает канал, соединяющий выходные и входные данные двух команд. Давайте отправим выходные данные сценария Python как входные данные в стандартную программу командной строки more, чтобы увидеть, как работает прием:

```
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
```

Здесь стандартный ввод teststreams снова поступает из файла, но вывод (посредством вызовов print) отправляется в другую программу, а не в файл или окно. Получающей программой является more — стандартная программа командной строки для постраничного просмотра, доступная на платформах Windows и Unix. Однако поскольку сценарии Python привязаны к стандартной потоковой модели, их можно использовать на обоих концах. Выходные данные одного сценария Python всегда можно передавать в качестве входных данных другому сценарию Python:

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

C:\...\PP4E\System\Streams> type reader.py
print('Got this: "%s"' % input())
     # Получено:
mport sys
data = sys.stdin.readline()[:-1]
print('The meaning of life is ', data, int(data) * 2)
     # Смысл жизни в том, что

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
```

На этот раз соединяются две программы на Python. Сценарий reader получает входные данные от сценария writer; оба сценария просто выполняют чтение и запись без учета механики потоков. На практике такое связывание представляет собой простую форму межпрограммного взаимодействия. Это облегчает *многократное использование* утилит, написанных для взаимодействия через stdin и stdout способами, которые мы никогда не ожидали. Скажем, программа на Python, которая сортирует текст из stdin, может быть применена к любому желаемому источнику данных, включая вывод других сценариев. Рассмотрим служебные сценарии командной строки Python в примерах 3.6 и 3.7, которые сортируют и суммируют строки в стандартном потоке ввода.

Пример 3.6. PP4E\System\Streams\sorter.py

```
import sys                                  # или sorted(sys.stdin)
lines = sys.stdin.readlines()               # сортировать входные строки из stdin
lines.sort()                                # и отправить результат в stdout
for line in lines: print(line, end='')      # для дальнейшей обработки
```

Пример 3.7. PP4E\System\Streams\adder.py

```
import sys
sum = 0
while True:
    try:
        line = input()                      # или sys.stdin.readlines()
    except EOFError:                        # или for line in sys.stdin:
        break                               # input отбрасывает \n в конце
    else:
        sum += int(line)
```

Такие инструменты общего назначения можно применять различными способами в командной строке для сортировки и суммирования произвольных файлов и выходных данных программы.

```
C:\...\PP4E\System\Streams> type data.txt
123
000
999
042
C:\...\PP4E\System\Streams> python sorter.py < data.txt     сортировать файл
000
042
123
999
C:\...\PP4E\System\Streams> python adder.py < data.txt      суммировать данные
                                                            в файле
1164
C:\...\PP4E\System\Streams> type data.txt | python adder.py суммировать данные
                                                            в выводе type
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                  сортировать данные
                                                            в выводе py
000
042
123
999
C:\...\PP4E\System\Streams> writer2.py | sorter.py          более короткая форма
...такой же вывод, как и в предыдущей команде в Windows...
C:\...\PP4E\System\Streams> python writer2.py | python sorter.py | python adder.py
1164
```

Последняя команда соединяет три сценария Python посредством стандартных потоков — вывод каждого предыдущего сценария передаются в качестве ввода следующему сценарию через синтаксис канала оболочки.

Альтернативные реализации сценариев `adder` и `sorter`

Есть несколько советов по кодированию: если вы внимательно посмотрите, то заметите, что сценарий `sorter.py` читает все данные `stdin` с помощью метода `readlines`, но `adder.py` читает по одной строке за раз. Когда источником ввода является другая программа, некоторые платформы запускают программы, соединенные каналами *параллельно*. В таких системах построчное чтение работает лучше, если передаваемые потоки данных велики, потому что инструментам чтения не нужно ждать окончания работы инструментов записи, чтобы заняться обработкой данных. Поскольку функция `input` просто осуществляет чтение из `stdin`, построчная схема, используемая `adder.py`, всегда может быть реализована посредством ручного чтения `sys.stdin`:

```
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)
```

В этой версии задействован тот факт, что тип `int` позволяет окружать цифры пробельными символами (метод `readline` возвращает строку, содержащую символы `\n`, но для их удаления с целью преобразования в `int` использовать `[:-1]` или `rstrip()` не придется). На самом деле для достижения того же эффекта можно применять более поздние файловые итераторы Python — скажем, цикл `for` автоматически захватывает по одной строке при проходе по файловому объекту (файловые итераторы рассматриваются в следующей главе):

```
C:\...\PP4E\System\Streams> type adder3.py
import sys
sum = 0
for line in sys.stdin: sum += int(line)
print(sum)
```

Однако изменение реализации `sorter` для построчного чтения может не дать большого прироста производительности, потому что метод `sort` списка требует, чтобы список уже был полным. Как будет показано в главе 18, написанные вручную алгоритмы сортировки обычно работают намного медленнее, чем метод сортировки списка в Python.

Интересно отметить, что эти два сценария также могут быть написаны гораздо более компактно в Python 2.4 и последующих версиях с использованием новой встроенной функции `sorted`, генераторных выражений и файловых итераторов. Представленные ниже варианты работают точно так же, как и оригиналы, но с заметно меньшим объемом исходного кода:

```
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))
```

В аргументе sum применяется генераторное выражение, которое очень похоже на списковое включение, но результаты возвращаются по одному, а не в физическом списке. Итогом оказывается оптимизация пространства. Дополнительные сведения ищите в книге *Изучаем Python* ("Диалектика", 2019 г.).

Перенаправление потоков и взаимодействие с пользователем

Ранее в разделе выходные данные teststreams.py передавались в стандартную программу командной строки more:

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

Но поскольку в предыдущей главе была реализована собственная утилита постраничного просмотра на Python, то почему бы ни настроить ее так, чтобы она принимала ввод и с потока stdin? Скажем, если изменить последние три строки в файле more.py из примера 2.1 в предыдущей главе...

```
if __name__== '__main__':       # в случае запуска, а не импортирования
  import sys
  if len(sys.argv) == 1:        # отобразить постранично stdin, если не указаны
                                # аргументы командной строки
      more(sys.stdin.read())
  else:
      more(open(sys.argv[1]).read())
```

...то, похоже, что появилась возможность перенаправления стандартного вывода teststreams.py в стандартный ввод 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
```

Для сценариев Python такой прием обычно работает. Здесь сценарий teststreams.py снова получает данные из файла. Как и в предыдущем разделе, вывод одной программы на Python направляется на вход другой — сценарию more.py в родительском каталоге (..).

Но в предыдущей команде с more.py скрывается тонкая проблема. На самом деле соединение в цепочку работало лишь по счастливой случайности: если бы вывод первого сценария оказался достаточно длинным для того, чтобы у пользователя запрашивалось продолжение, то сценарий more.py потерпел бы неудачу (особенно, когда input при взаимодействии с пользователем вызовет ошибку EOFError).

Проблема в том, что дополненный файл more.py использует stdin для двух разных целей. Он читает ответ от интерактивного пользователя из stdin, вызывая input, но теперь *также* принимает основной вводимый текст из stdin. Когда поток stdin действительно перенаправляется во входной файл или канал, его невозможно применять для ввода ответа от интерактивного пользователя; он содержит только текст источника ввода. Более того, т.к. поток stdin перенаправляется еще до запуска программы, невозможно узнать, что он означает, до перенаправления в командной строке.

В случае приема входных данных из stdin *и* применения консоли для взаимодействия с пользователем пришлось проделать дополнительную работу — использовать специальные интерфейсы для прямого чтения ответов пользователя с клавиатуры вместо стандартного ввода. В Windows такие инструменты есть в стандартной библиотеке msvcrt; на многих Unix-подобных платформах обычно достаточно чтения из устройства /dev/tty.

Поскольку, возможно, это неясный вариант использования, мы делегируем полное решение предложенному ниже упражнению. В примере 3.8 приведена модифицированная версия сценария more только для Windows, которая постранично отображает содержимое стандартного входного потока, когда вызывается без аргументов, но также использует низкоуровневые и специфичные для платформы инструменты для взаимодействия с пользователем через клавиатуру, если возникает такая необходимость.

Пример 3.8. PP4E\System\Streams\moreplus.py

```
"""
Разделяет и интерактивно постранично выводит строку, файл или поток текста в stdout;
при запуске в качестве сценария постранично выводит содержимое stdin или файла,
имя которого передается в командной строке; если ввод производится из stdin,
то его нельзя применять для ответа пользователя - используйте инструменты
для конкретной платформы или графический интерфейс.
"""

import sys

def getreply():
    """
    читает клавишу, нажатую интерактивным пользователем,
    даже если поток stdin перенаправлен в файл или канал
    """
    if sys.stdin.isatty():           # если поток stdin является консолью,
        return input('?')            # читать строку ответа из stdin
    else:
        if sys.platform[:3] == 'win':# если поток stdin был перенаправлен,
            import msvcrt            # его нельзя использовать для чтения
                                     # ответа от пользователя
            msvcrt.putch(b'?')
            key = msvcrt.getche()    # использовать инструменты консоли Windows
            msvcrt.putch(b'\n')   # getch() не выводит символ для нажатой клавиши
        else:
            assert False, 'platform not supported'  # платформа не поддерживается
            #linux?: open('/dev/tty').readline()[:-1]

def more(text, numlines=10):
    """
```

```
    постранично выводит многострочную строку в 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', 'y', 'Y']: break
if __name__=='__main__':        # при запуске, но не импортировании
    if len(sys.argv) == 1:      # если аргументы командной строки отсутствуют,
        more(sys.stdin.read())  # тогда постранично выводить содержимое stdin,
                                # а не введенные данные
    else:
        more(open(sys.argv[1]).read()) # в противном случае постранично выводить
                                       # содержимое аргумента с именем файла
```

Большинство нового кода в данной версии находится в функции `getreply`. Метод `isatty` файла сообщает, подключен ли поток `stdin` к консоли; если это так, то мы просто читаем ответы из `stdin`, как делали ранее. Конечно, такая дополнительная логика должна добавляться только к сценариям, которые предназначены для взаимодействия с пользователями консоли *и* получения ввода из `stdin`. Скажем, в приложении с графическим пользовательским интерфейсом взамен можно было бы отображать всплывающие диалоговые окна, привязывать события нажатия клавиш на клавиатуре для запуска обратных вызовов и т.д. (вы ознакомитесь с графическими пользовательскими интерфейсами в главе 7).

Однако, имея в своем распоряжении многократно используемую функцию `getreply`, мы можем безопасно запускать утилиту `moreplus` разнообразными способами. Как и раньше, мы можем напрямую импортировать и вызывать функцию этого модуля, передавая любую строку, которую необходимо отобразить:

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

Кроме того, при запуске с *аргументом* командной строки сценарий `moreplus.py` интерактивно постранично выводит файла с указанным именем:

```
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
"""
разделяет и интерактивно постранично выводит строку, файл или поток текста в stdout;
при запуске в качестве сценария постранично выводит содержимое stdin или файла,
имя которого передается в командной строке; если ввод производится из stdin,
то его нельзя применять для ответа пользователя - используйте инструменты
для конкретной платформы или графический интерфейс.
"""

import sys

def getreply():
?n
```

Но теперь сценарий также корректно постранично выводит текст, перенаправленный в stdin из *файла* либо по каналу, даже если текст слишком длинный, чтобы уместиться в один отображаемый блок. В большинстве оболочек такой ввод отправляется через перенаправление или операции канала, например:

```
C:\...\PP4E\System\Streams> python moreplus.py < moreplus.py
"""
Разделяет и интерактивно постранично выводит строку, файл или поток текста в stdout;
при запуске в качестве сценария постранично выводит содержимое stdin или файла,
имя которого передается в командной строке; если ввод производится из stdin,
то его нельзя применять для ответа пользователя - используйте инструменты
для конкретной платформы или графический интерфейс.
"""

import sys

def getreply():
?n

C:\...\PP4E\System\Streams> type moreplus.py | python moreplus.py
"""
Разделяет и интерактивно постранично выводит строку, файл или поток текста в stdout;
при запуске в качестве сценария постранично выводит содержимое stdin или файла,
имя которого передается в командной строке; если ввод производится из stdin,
то его нельзя применять для ответа пользователя - используйте инструменты
для конкретной платформы или графический интерфейс.
"""

import sys

def getreply():
?n
```

Наконец, передача вывода одного сценария Python на его вход теперь работает ожидаемым образом, не нарушая взаимодействие с пользователем (а не только потому, что нам повезло):

```
......\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
```

Здесь стандартный *вывод* одного сценария Python подается на стандартный *ввод* другого сценария Python, расположенного в том же каталоге: moreplus.py читает вывод teststreams.py.

Все перенаправления в таких командных строках работают лишь потому, что сценариям безразлично, чем является стандартный ввод и вывод — интерактивными пользователями, файлами или каналами между программами. Например, при запуске в качестве сценария moreplus.py просто читает поток sys.stdin; оболочка командной строки (вроде DOS в Windows или csh в Linux) перед запуском сценария присоединяет такие потоки к источнику, подразумеваемому командной строкой. Для доступа к этим источникам в сценариях вне зависимости от их истинной природы применяются предварительно открытые файловые объекты stdin и stdout.

Следует отметить, что мы запускали данный сценарий постраничного вывода четырьмя способами: импортированием и вызовом его функции, передачей аргумента командной строки с именем файла, перенаправлением stdin в файл и передачей вывода команды по каналу в stdin. Благодаря поддержке импортируемых функций, аргументов командной строки и стандартных потоков код системных инструментов Python можно многократно использовать в самых разных режимах.

Перенаправление потоков в объекты Python

Все предыдущие перенаправления стандартных потоков работают для программ, написанных на любом языке, которые подключаются к стандартным потокам и больше полагаются на процессор командной строки оболочки, чем на сам Python. Синтаксис перенаправления командной строки, такой как < имя_файла и | программа, обрабатывается оболочкой, а не интерпретатором Python. Больше присущая Python форма перенаправления может быть реализована внутри самих сценариев путем сброса sys.stdin и sys.stdout в файловые объекты. Главный трюк, лежащий в основе этого режима, заключается в том, что все, выглядящее как файл с точки зрения методов, будет работать как стандартный поток в Python. Имеет значение только интерфейс объекта (иногда называемый его протоколом), а не конкретный тип данных объекта.

- Потоку sys.stdin можно присваивать любой объект, предоставляющий файловые методы *чтения*, чтобы ввод поступал из методов чтения этого объекта.
- Потоку sys.stdout можно присваивать любой объект, предоставляющий файловые методы *записи*, в результате чего весь стандартный вывод будет отправлен методам этого объекта.

Поскольку функции print и input просто вызывают методы write и readline любых объектов, на которые ссылаются sys.stdout и sys.stdin, мы можем применять описанный прием как для предоставления, так и для перехвата стандартного потокового текста с помощью объектов, реализованных в виде классов.

Если вы уже изучали Python, то наверняка знаете, что подобная совместимость обычно называется *полиморфизмом* — абсолютно не важно, что представляет собой объект, и не имеет значения, что делает его интерфейс, лишь бы он обеспечивал ожидаемый интерфейс. Такой либеральный подход к типам данных во многом объясняет лаконичность и гибкость кода на Python. Здесь он предоставляет сценариям возможность сбрасывания собственных потоков. В примере 3.9 представлен код служебного модуля, где демонстрируется данная концепция.

Пример 3.9. PP4E\System\Streams\redirect.py

```
"""
Файловые объекты, которые сохраняют стандартный выходной текст в строке
и предоставляют стандартный входной текст из строки; redirect запускает переданную
функцию с ее выходными и входными потоками, сбрасываемыми в эти файловые объекты
"""
```

```
import sys                                      # получить встроенные модули
class Output:                                   # эмулировать выходной файл
    def  init (self):
        self.text = ''                          # при создании строка пуста
    def write(self, string):                    # добавить строку байтов
        self.text += string
    def writelines(self, lines):                # добавить все строки в список
        for line in lines: self.write(line)
class Input:    # simulated input file
    def  init (self, input=''):                 # стандартный аргумент
        self.text = input                       # при создании сохранить строку
    def read(self, size=None):                  # необязательный аргумент
        if size == None:                        # прочитать N байтов или все
            res, self.text = self.text, ''
        else:
            res, self.text = self.text[:size],
            self.text[size:]
        return res
    def readline(self):
        eoln = self.text.find('\n') # найти смещение следующего символа конца строки
        if eoln == -1:              # извлечь строку до символа конца строки
            res, self.text = self.text, ''
        else:
            res, self.text = self.text[:eoln+1], self.text[eoln+1:]
        return res
def redirect(function, pargs, kargs, input):  # перенаправить stdin/stdout
    savestreams = sys.stdin, sys.stdout       # запустить объект функции
    sys.stdin = Input(input)                  # возвратить текст из stdout
    sys.stdout = Output()
    try:
        result = function(*pargs, **kargs)    # запустить функцию с аргументами
        output = sys.stdout.text
    finally:
        sys.stdin, sys.stdout = savestreams   # восстановить независимо
                                              # от наличия исключения
        return (result, output)               # возвратить результат, если
                                              # не было исключений
```

В модуле определены два класса, которые маскируются под настоящие файлы.

- `Output`. Определяет интерфейс (протокол) метода записи, ожидаемый от выходных файлов, но сохраняет весь записываемый вывод в строке, расположенной в памяти.
- `Input`. Определяет интерфейс, ожидаемый от входных файлов, но обеспечивает ввод по запросу из строки в памяти, переданной во время создания объекта.

Функция `redirect` в конце файла объединяет два вышеупомянутых объекта для запуска одной функции с вводом и выводом, полностью перенаправляемым на объекты Python. Передаваемая функция для запуска не должна знать или заботиться о том, чтобы вызовы в ней функций `print` и `input`, а также вызовы методов `stdin` и `stdout` обращались к классу, а не к реальному файлу, каналу или пользователю.

В целях демонстрации импортируем и запустим функцию interact из сценария teststreams (см. пример 3.5), который мы запускали в оболочке (для использования функции перенаправления нужно иметь дело с функциями, а не с файлами). В случае прямого запуска функция осуществляет чтение с клавиатуры и вывод на экран, как если бы она запускалась в виде программы без перенаправления:

```
C:\...\PP4E\System\Streams> python
>>> from teststreams import interact
>>> interact()
Hello stream world
Enter a number>2
2 squared is 4
Enter a number>3
3 squared is 9
Enter a number>^Z
Bye
>>>
```

Теперь давайте запустим эту функцию под управлением функции перенаправления в redirect.py и передадим ей готовый входной текст. В таком режиме функция interact получает входные данные из указанной строки ('4\n5\n6\n' — три строки с явными символами конца строки), а результатом выполнения функции является кортеж, который содержит ее возвращаемое значение и строку с текстом, записанным в стандартный поток вывода:

```
>>> from redirect import redirect
>>> (result, output) = redirect(interact, (), {}, '4\n5\n6\n')
>>> print(result)
None
>>> output
'Hello stream world\nEnter a number>4 squared is 16\nEnter a number>5 после
возведения в квадрат равно 25\nEnter a number>6 squared is 36\nEnter a
number>Bye\n'
```

Вывод представляет собой одну длинную строку со всем текстом, записанным в стандартный вывод. Он будет выглядеть лучше, если мы передадим его функции print, разделяя с помощью метода splitlines строкового объекта:

```
>>> for line in output.splitlines(): print(line)
...
Hello stream world
Enter a number>4 squared is 16
Enter a number>5 squared is 25
Enter a number>6 squared is 36
Enter a number>Bye
```

Более того, мы можем повторно использовать модуль more.py, реализованный в предыдущей главе (см. пример 2.1); при импортировании между каталогами предполагается, что каталог, содержащий корень PP4E, находится в пути поиска модулей (при необходимости измените настройку PYTHONPATH):

```
>>> from PP4E.System.more import more
>>> more(output)
```

```
Hello stream world
Enter a number>4 squared is 16
Enter a number>5 squared is 25
Enter a number>6 squared is 36
Enter a number>Bye
```

Конечно, это искусственный пример, но проиллюстрированные в нем приемы широко применимы. Например, к программе, написанной для взаимодействия с пользователем командной строки, несложно добавить графический пользовательский интерфейс. Нужно лишь перехватить стандартный вывод с помощью объекта, такого как описанный ранее экземпляр класса `Output`, и вывести текстовую строку в окно. Аналогичным образом стандартный ввод может быть перенаправлен в объект, который извлекает текст из графического интерфейса (скажем, из всплывающего диалогового окна). Поскольку классы позволяют подключаться к реальным файлам, мы можем использовать их в любом инструменте, который ожидает файл. Обратите внимание на модуль `guiStreams` для перенаправления потоков в графическом пользовательском интерфейсе из главы 10, который обеспечивает конкретную реализацию нескольких идей подобного рода.

Служебные классы `io.StringIO` и `io.BytesIO`

Прием с перенаправлением потоков в объекты из предыдущего раздела оказался настолько удобным, что теперь стандартный библиотечный модуль автоматизирует задачу для многих случаев применения (хотя некоторые случаи использования, такие как графические пользовательские интерфейсы, могут по-прежнему требовать написания дополнительного кода). Инструмент стандартной библиотеки предоставляет объект, который отображает интерфейс файлового объекта на строки в памяти и из них. Например:

```
>>> from io import StringIO
>>> buff = StringIO()                    # сохранить записанный текст в строку
>>> buff.write('spam\n')
5
>>> buff.write('eggs\n')
5
>>> buff.getvalue()
'spam\neggs\n'
>>> buff = StringIO('ham\nspam\n')       # обеспечить ввод из строки
>>> buff.readline()
'ham\n'
>>> buff.readline()
'spam\n'
>>> buff.readline()
''
```

Как и в предыдущем разделе, экземпляры объектов `StringIO` можно присваивать `sys.stdin` и `sys.stdout` с целью перенаправления потоков для вызовов `input` и `print`, а также передавать в любой код, который ожидает реального файлового объекта. Как обычно, правила игры в Python определяются *интерфейсом* объекта, а не конкретным типом данных:

```
>>> from io import StringIO
>>> import sys
>>> buff = StringIO()

>>> temp = sys.stdout
>>> sys.stdout = buff
>>> print(42, 'spam', 3.141)            # или print(..., file=buff)

>>> sys.stdout = temp                   # восстановить исходный поток
>>> buff.getvalue()
'42 spam 3.141\n'
```

Следует отметить, что существует класс `io.BytesIO` с аналогичным поведением, но он отображает файловые операции на буфер байтов в памяти, а не на строку `str`:

```
>>> from io import BytesIO
>>> stream = BytesIO()
>>> stream.write(b'spam')
>>> stream.getvalue()
b'spam'

>>> stream = BytesIO(b'dpam')
>>> stream.read()
b'dpam'
```

Из-за значительных различий между текстовыми и двоичными данными в Python 3.X такой вариант может лучше подходить для сценариев, работающих с двоичными данными. Разница между текстовыми и двоичными файлами обсуждается в следующей главе.

Захват потока `stderr`

Ранее внимание было сосредоточено на перенаправлении `stdin` и `stdout`, но `stderr` можно аналогичным образом перенаправлять в файлы, каналы и объекты. В то время как перенаправление `stderr` поддерживается в некоторых оболочках, его также просто реализовать в сценарии Python. Например, присваивание `sys.stderr` экземпляру класса вроде `Output` или объекту `StringIO` в примере из предыдущего раздела позволяет сценарию перехватывать текст, записываемый в `stderr`.

Сам интерпретатор Python использует поток `stderr` для вывода текста сообщений об ошибках (графический пользовательский интерфейс IDLE перехватывает его текст и по умолчанию окрашивает в красный цвет). Тем не менее, никакие инструменты более высокого уровня для стандартного потока ошибок не делают то, что функции `print` и `input` обеспечивают для выходного и входного потоков. Если вы хотите вывести что-либо в поток `stderr`, то понадобится вызвать `sys.stderr.write()` явно или прочитать следующий раздел, чтобы узнать о трюке с вызовом `print`, который упрощает эту задачу.

Перенаправление потока стандартных ошибок из командной строки оболочки немного сложнее и менее переносимо. В большинстве Unix-подобных систем вывод `stderr` обычно можно перехватывать с применением синтаксиса перенаправления оболочки в форме `команда > вывод 2>&1`. Однако такой синтаксис может не работать на некоторых платформах и даже отличаться в зависимости от оболочки Unix; для получения более подробной информации обратитесь к страницам справочного руководства по вашей оболочке.

Синтаксис перенаправления в вызовах print

Поскольку переустановка атрибутов потоков в новые объекты оказалась настолько популярной, встроенная функция print также была расширена возможностью явного включения файла, куда должны быть отправлены выходные данные. Оператор следующего вида:

```
print(stuff, file=afile) # afile является объектом, а не именем строковой переменной
```

выводит данные в файл, а не в sys.stdout. Результат подобен простому присваиванию sys.stdout объекту, но отсутствует необходимость в сохранении и восстановлении для возврата к исходному выходному потоку (как показано в разделе, посвященном перенаправлению потоков на объекты). Например, показанные ниже операторы:

```
import sys
print('spam' * 2, file=sys.stderr)
```

обеспечат отправку текста в стандартный поток ошибок, а не в sys.stdout, но только на время одного вызова print. Приведенный далее нормальный оператор print (без file) выводит в sys.stdout обычным образом. Точно так же в качестве выходного файла можно применять либо специальный класс, либо класс стандартной библиотеки:

```
>>> from io import StringIO
>>> buff = StringIO()
>>> print(42, file=buff)
>>> print('spam', file=buff)
>>> print(buff.getvalue())
42
spam
>>> from redirect import Output
>>> buff = Output()
>>> print(43, file=buff)
>>> print('eggs', file=buff)
>>> print(buff.text)
43
eggs
```

Другие варианты перенаправления: еще раз о функции os.popen и модуле subprocess

Ближе к концу предыдущей главы была впервые представлена встроенная функция os.popen и ее аналог subprocess.Popen, которые обеспечивают возможность перенаправления потоков другой команды из программы на Python. Вы видели, что такие инструменты можно использовать для запуска командной строки оболочки (строка, которая обычно вводится в окне командной строки DOS или csh), но также предоставления файлового объекта Python, подключенного к выходному потоку команды — чтение из файлового объекта позволяет сценарию принимать вывод другой программы. Затем было выдвинуто предположение, что упомянутые инструменты можно применять и для подключения к входным потокам.

По этой причине инструменты `os.popen` и `subprocess` являются еще одним способом перенаправления потоков порожденных программ и близкими родственниками ряда методов, с которыми вы только что ознакомились. Их эффект очень похож на синтаксис организации каналов `|` в командной строке для перенаправления потоков в программы (их названия означают "pipe open" — "открыть канал"), но они запускаются внутри сценария и предоставляют файловый интерфейс для потоков, связанных каналом. По духу они аналогичны функции перенаправления, но основаны на запуске программ (не на вызове функций), и потоки команды обрабатываются в порождающем сценарии как файлы (не привязанные к объектам класса). Указанные инструменты перенаправляют потоки программы, запускаемой сценарием, а не потоки самого сценария.

Перенаправление ввода и вывода с помощью `os.popen`

На самом деле при передаче флага нужного режима мы перенаправляем потоки вывода *или* ввода порожденной программы в файл в вызывающих сценариях и можем получить код состояния выхода порожденной программы из метода `close` (`None` здесь означает отсутствие ошибок). В качестве иллюстрации рассмотрим следующие два сценария:

```
C:\...\PP4E\System\Streams> type hello-out.py
print('Hello shell world')

C:\...\PP4E\System\Streams> type hello-in.py
inp = input()
open('hello-in.txt', 'w').write('Hello ' + inp + '\n')
```

Эти сценарии можно запускать из окна системной оболочки обычным образом:

```
C:\...\PP4E\System\Streams> python hello-out.py
Hello shell world

C:\...\PP4E\System\Streams> python hello-in.py
Brian

C:\...\PP4E\System\Streams> type hello-in.txt
Hello Brian
```

Как было показано в предыдущей главе, сценарии Python также могут читать *вывод* из других программ и сценариев такого рода, используя следующий код:

```
C:\...\PP4E\System\Streams> python
>>> import os
>>> pipe = os.popen('python hello-out.py')  # режим r принимается
                                            # по умолчанию -- чтение stdout
>>> pipe.read()
'Hello shell world\n'
>>> print(pipe.close())            # состояние завершения: None -- нормально
None
```

Но сценарии Python также могут предоставлять *ввод* для стандартных входных потоков порожденных программ — передача аргумента режима `w` вместо стандартного `r` соединяет возвращаемый объект с входным потоком порожденной программы. То, что записывается в порожденной программе, отображается как ввод в запущенной программе:

```
>>> pipe = os.popen('python hello-in.py', 'w')  # w -- запись в
                                                # поток stdin программы
>>> pipe.write('Gumby\n')
6
>>> pipe.close()                    # символ \n в конце является необязательным
>>> open('hello-in.txt').read()     # вывод отправляется в файл
'Hello Gumby\n'
```

Функция `popen` также способна запускать командную строку в виде независимого процесса на платформах, поддерживающих такое понятие. Она принимает необязательный третий аргумент, который можно использовать для управления буферизацией записанного текста, которую мы здесь усовершенствуем.

Перенаправление ввода и вывода с помощью subprocess

Для еще большего контроля над потоками порожденных программ можно применять модуль `subprocess`, который был представлен в предыдущей главе. Ранее вы узнали, что данный модуль способен эмулировать функциональность `os.popen` и вдобавок выполнять такие функции, как двунаправленные потоковые коммуникации (доступ к вводу и выводу программы), плюс связывание вывода одной программы с вводом другой.

Например, модуль `subprocess` предлагает несколько способов запуска программы и получения ее стандартного вывода и состояния завершения. Ниже демонстрируются три распространенных способа использования этого модуля для запуска программы и перенаправления ее потока *вывода* (вспомните из главы 2, что может понадобиться передать в `Popen` и `call` аргумент `shell=True`, чтобы приведенные здесь примеры работали на Unix-подобных платформах):

```
C:\...\PP4E\System\Streams> python
>>> from subprocess import Popen, PIPE, call
>>> X = call('python hello-out.py')                          # удобство
Hello shell world
>>> X
0

>>> pipe = Popen('python hello-out.py', stdout=PIPE)
>>> pipe.communicate()[0]                                    # (stdout, stderr)
b'Hello shell world\r\n'
>>> pipe.returncode                                          # код завершения
0

>>> pipe = Popen('python hello-out.py', stdout=PIPE)
>>> pipe.stdout.read()
b'Hello shell world\r\n'
>>> pipe.wait()                                              # код завершения
0
```

Вызов `call` в первом из трех показанных способов представляет собой просто удобную функцию (есть и другие, которые вы можете найти в руководстве по библиотеке Python), а функция `communicate` во втором — подобного рода удобство для третьего способа (она отправляет данные в `stdin`, читает данные из `stdout` вплоть до достижения конца файла и ожидает завершения процесса).

Перенаправление и подключение к *входному* потоку порожденной программы реализуется почти так же просто, хотя и немного сложнее, чем подход с вызовом `os.popen` для файлового режима `'w'`, описанный в предыдущем разделе (как упоминалось в предыдущей главе, метод `os.popen` реализован с помощью модуля `subprocess` и потому сам по себе сегодня является чем-то вроде удобной функции):

```
>>> pipe = Popen('python hello-in.py', stdin=PIPE)
>>> pipe.stdin.write(b'Pokey\n')
6
>>> pipe.stdin.close()
>>> pipe.wait()
0
>>> open('hello-in.txt').read()                  # вывод отправляется в файл
'Hello Pokey\n'
```

На самом деле посредством модуля `subprocess` мы можем организовать получение *и входных, и выходных* потоков порожденной программы. Давайте в целях демонстрации повторно воспользуемся простыми сценариями записи и чтения, которые были реализованы ранее:

```
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)
```

Код, подобный представленному ниже, может как читать, так и записывать в сценарий `reader.py` — объект канала имеет два файловых объекта, доступных в качестве присоединенных атрибутов, один из которых подключается к входному потоку, а другой — к выходному (пользователи Python 2.X могут распознать их как эквивалент кортежа, возвращаемого ныне несуществующей функцией `os.popen2`):

```
>>> pipe = Popen('python reader.py', stdin=PIPE, stdout=PIPE)
>>> pipe.stdin.write(b'Lumberjack\n')
11
>>> pipe.stdin.write(b'12\n')
3
>>> pipe.stdin.close()
>>> output = pipe.stdout.read()
>>> pipe.wait()
0
>>> output
b'Got this: "Lumberjack"\r\nThe meaning of life is 12 24\r\n'
```

В главе 5 вы узнаете о том, что при взаимодействии с программной подобного рода следует проявлять осторожность; буферизированные выходные потоки могут привести к взаимоблокировке, если операции записи и чтения чередуются, и вполне вероятно в качестве обходного пути придется рассмотреть применение таких инструментов, как утилита Pexpect (рассматривается позже).

Наконец, возможно еще более экзотическое управление потоком — следующие команды *соединяют две программы*, перенаправляя вывод одного сценария Python в другой, сначала с помощью синтаксиса оболочки, а затем с помощью модуля subprocess:

```
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
C:\...\PP4E\System\Streams> python
>>> from subprocess import Popen, PIPE
>>> p1 = Popen('python writer.py', stdout=PIPE)
>>> p2 = Popen('python reader.py', stdin=p1.stdout, stdout=PIPE)
>>> output = p2.communicate()[0]
>>> output
b'Got this: "Help! Help! I'm being repressed!"\r\nThe meaning of life is 42 84\r\n'
>>> p2.returncode
0
```

Похожее решение можно получить с помощью функции os.popen, но тот факт, что ее каналы доступны для чтения или записи (а не для того и другого), не позволяет перехватить вывод второго сценария в коде:

```
>>> import os
>>> p1 = os.popen('python writer.py', 'r')
>>> p2 = os.popen('python reader.py', 'w')
>>> p2.write( p1.read() )
36
>>> X = p2.close()
Got this: "Help! Help! I'm being repressed!"
The meaning of life is 42 84
>>> print(X)
None
```

С более широкой точки зрения вызов os.popen и модуль subprocess являются переносимыми эквивалентами Python синтаксиса Unix-подобной оболочки для перенаправления потоков порожденных программ. Однако версии Python также работают в Windows и представляют собой наиболее независимый от платформы способ запуска другой программы из сценария Python. Командные строки, которые вы им передаете, могут различаться в зависимости от платформы (например, для списка каталогов требуется ls в Unix и dir в Windows), но сам вызов работает на всех основных платформах, поддерживаемых Python.

На Unix-подобных платформах комбинация вызовов os.fork, os.pipe, os.dup и некоторых вариантов os.exec также может использоваться для запуска новой независимой программы с потоками, связанными с потоками родительской программы. Таким образом, это еще один способ перенаправления потоков и низкоуровневый эквивалент инструментов вроде os.popen (os.fork доступен в Cygwin Python для Windows).

Тем не менее, поскольку все они представляют собой более сложные инструменты параллельной обработки, мы отложим дальнейшие подробности до главы 5, особенно в том, что касается каналов и кодов состояния завершения. В главе 6 мы снова возвратимся к исследованию модуля subprocess, чтобы реализовать механизм регрессионного тестирования, который перехватывает все *три* стандартных потока порождаемых тестовых сценариев — потоки ввода, вывода и ошибок.

Но сначала в главе 4 мы продолжим обзор системных интерфейсов Python рассмотрением инструментов, которые доступны для обработки файлов и каталогов. Хотя мы немного сместим центр внимания, вы обнаружите, что кое-что из того, что было описано здесь, пригодится в качестве общих инструментов, связанных с системой. Порождение команд оболочки, например, предоставляет способы проверки каталогов, а файловый интерфейс, который обсуждается в следующей главе, лежит в основе методов потоковой обработки, которые были исследованы в настоящей главе.

Сравнение Python и csh

Если вы знакомы с другими распространенными языками сценариев оболочки, то может быть полезно сравнить их с Python. Ниже показан простой сценарий на языке оболочки Unix под названием csh, который отправляет по электронной почте все файлы с суффиксом `.py` (т.е. все файлы с исходным кодом Python) из текущего рабочего каталога на фиктивный адрес:

```
#!/bin/csh
foreach x (*.py)
    echo $x
    mail eric@halfabee.com -s $x < $x
end
```

Эквивалентный сценарий Python выглядит похоже, но чуть длиннее:

```
#!/usr/bin/python
import os, glob
for x in glob.glob('*.py'):
    print(x)
    os.system('mail eric@halfabee.com -s %s < %s' % (x, x))
```

Так как язык Python в отличие от csh предназначен не только для написания сценариев оболочки, системные интерфейсы необходимо импортировать и вызывать явно. А поскольку Python — не просто язык для обработки строк, строки символов должны заключаться в кавычки, как в C.

Хотя в таком простом сценарии может потребоваться дополнительный набор на клавиатуре, за счет того, что Python является языком общего назначения, он становится лучшим инструментом, когда мы покидаем область тривиальных программ. Скажем, предыдущий сценарий можно было бы расширить для передачи файлов по FTP, открывать окно выбора сообщений, отображать состояние, извлекать сообщения из данных SQL и применять COM-объекты в Windows, причем все это с использованием стандартных инструментов Python.

Сценарии Python также обычно более переносимы на другие платформы по сравнению со сценариями csh. Например, в случае применения интерфейсного модуля Python SMTP для отправки сообщений электронной почты вместо инструмента командной строки `mail` из Unix сценарий мог бы выполняться на любой машине с интерпретатором Python и подключением к Интернету (как будет показано в главе 13, для SMTP требуются только сокеты). И подобно C не нужно использовать `$` для получения значений переменных; что еще можно было бы ожидать от свободного языка?