ЧАСТЬ III

Операторы и синтаксис

ГЛАВА 10

Введение в операторы Python

Теперь, когда вы знакомы с основными встроенными типами объектов Python, мы приступаем к исследованию главных форм операторов Python. Как и в предыдущей части, мы начнем с обобщенного введения в синтаксис операторов, после чего в последующих нескольких главах более подробно рассмотрим специфические операторы.

Попросту говоря, *операторы* — это то, что вы пишете для сообщения Python о том, что должны делать ваши программы. Если согласно главе 4 программы "делают дела с помощью оснащения", тогда операторы являются способом указания вида *дел*, которые делает программа. Более формально Python представляет собой процедурный, основанный на операторах язык; за счет комбинирования операторов вы задаете *процедуру*, которую Python выполняет для достижения целей программы.

Еще раз о концептуальной иерархии Python

Другой способ постижения роли операторов предусматривает мысленный возврат к концептуальной иерархии, введенной в главе 4, где речь шла о встроенных объектах и выражениях, используемых для манипулирования ими. В настоящей главе иерархия поднимается на следующий уровень структуры программ Python.

1. Программы состоят из модулей.
2. Модули содержат операторы.
3. Операторы содержат выражения.
4. Выражения создают и обрабатывают объекты.

В своей основе программы, написанные на языке Python, состоят из операторов и выражений. Выражения обрабатывают объекты и встраиваются в операторы. Операторы кодируют более крупную *логику* работы программы — они применяют и направляют выражения для обработки объектов, которые вы изучали в предшествующих главах. Более того, операторы представляют собой место, где появляются объекты (например, в выражениях внутри операторов присваивания), а некоторые операторы создают совершенно новые виды объектов (функции, классы и т.д.). На верхнем уровне операторы всегда существуют в модулях, которые сами управляются с помощью операторов.

Операторы Python

В табл. 10.1 приведена сводка по набору операторов Python. Каждый оператор Python имеет собственное специфическое назначение и собственный специфический *синтаксис* — правила, определяющие его структуру, — хотя, как мы увидим, многие разделяют общие синтаксические шаблоны, а роли ряда операторов перекрываются. В табл. 10.1 также даны примеры каждого оператора, закодированные в соответствии с его синтаксическими правилами. В ваших программах такие единицы кода могут выполнять действия, повторять задачи, делать выбор, создавать более крупные программные структуры и т.д.

В настоящей части книги обсуждаются операторы с первой строки таблицы вплоть до `break` и `continue`. Вам уже неформально было представлено несколько операторов из табл. 10.1; в этой части книги мы восполним опущенные ранее детали, ознакомим с остатком набора процедурных операторов Python и раскроем полную синтаксическую модель. Операторы, расположенные ниже в табл. 10.1, которые касаются более крупных программных единиц — функций, классов, модулей и исключений — подводят к более крупным понятиям программирования, так что каждый из них будет рассматриваться в отдельном разделе. Более специализированные операторы (вроде `del`, который удаляет разнообразные компоненты) раскрываются в других местах книги, а также описаны в стандартных руководствах по Python.

Таблица 10.1. Операторы Python

Оператор	Роль	Пример
Присваивания	Создание ссылок	`a, b = 'good', 'bad'`
Вызовы и другие выражения	Выполнение функций	`log.write("spam, ham")`
Вызовы `print`	Вывод объектов	`print('The Killer', joke)`
`if/elif/else`	Выбор действий	`if "python" in text:` `  print(text)`
`for/else`	Итерация	`for x in mylist:` `  print(x)`
`while/else`	Универсальные циклы	`while X > Y:` `  print('hello')`
`pass`	Пустой заполнитель	`while True:` `  pass`
`break`	Выход из цикла	`while True:` `  if exittest(): break`
`continue`	Продолжение цикла	`while True:` `  if skiptest(): continue`
`def`	Функции и методы	`def f(a, b, c=1, *d):` `  print(a+b+c+d[0])`
`return`	Результаты функций	`def f(a, b, c=1, *d):` `  return a+b+c+d[0]`
`yield`	Генераторные функции	`def gen(n):` `  for i in n: yield i*2`

Окончание табл. 10.1

Оператор	Роль	Пример

`global`	Пространства имен	`x = 'old'`
		`def function():`
		`  global x, y; x = 'new'`

`nonlocal`	Пространства имен (Python 3.X)	`def outer():`
		`  x = 'old'`
		`  def function():`
		`  nonlocal x; x = 'new'`

`import`	Доступ к модулям	`import sys`
`from`	Доступ к атрибутам	`from sys import stdin`

`class`	Построение объектов	`class Subclass(Superclass):`
		`  staticData = []`
		`  def method(self): pass`

`try/except/finally`	Перехват исключений	`try:`
		`  action()`
		`except:`
		`  print('action error')`

`raise`	Генерация исключений	`raise EndSearch(location)`
`assert`	Отладочные проверки	`assert X > Y, 'X too small'`
`with/as`	Диспетчеры контекста (Python 3.X, 2.6+)	`with open('data') as myfile:` `  process(myfile)`

`del`	Удаление ссылок	`del data[k]`
		`del data[i:j]`
		`del obj.attr`
		`del variable`

Формально в табл. 10.1 воспроизведены операторы Python 3.X. Хотя этого перечня операторов достаточно для обзора и справочных целей, в том виде, как есть, он не совсем полон. Ниже описано несколько тонкостей относительно его содержимого.

- Операторы присваивания имеют различные синтаксические формы, описанные в главе 11: базовая, последовательности, дополненная и др.
- Формально `print` в Python 3.X не является ни зарезервированным словом, ни оператором, а вызовом встроенной функции; но поскольку `print` почти всегда будет выполняться в виде оператора с выражением (и часто занимать отдельную строку), такой вызов в общем случае трактуется как разновидность оператора. Операции вывода рассматриваются в главе 11.
- Начиная с Python 2.5, `yield` — также выражение, а не оператор; подобно `print` оно обычно используется как оператор с выражением и потому включено в табл. 10.1, но в главе 20 мы увидим, что сценарии иногда присваивают или по-другому применяют его результат. Будучи выражением, `yield` также является ключевым словом в отличие от `print`.

Бо́льшая часть табл. 10.1 применима и к Python 2.X за исключением случаев, когда это не так — если вы используете Python 2.X, то ниже приведено несколько замечаний, которые его касаются.

- В Python 2.X оператор `nonlocal` недоступен; как будет показано в главе 17, существуют альтернативные способы достижения того же эффекта сохранения перезаписываемого состояния.
- В Python 2.X `print` представляет собой оператор, а не вызов встроенной функции, со специфическим синтаксисом, раскрываемым в главе 11.
- В Python 2.X встроенная функция выполнения кода `exec` из Python 3.X является оператором со специфическим синтаксисом; однако поскольку он поддерживает окружающие круглые скобки, вы можете применять в коде Python 2.X форму вызова `exec`, принятую в Python 3.X.
- В Python 2.5 операторы `try/except` и `try/finally` были объединены: раньше они представляли собой два отдельных оператора, но теперь `except` и `finally` можно записывать в том же самом операторе `try`.
- В Python 2.5 оператор `with/as` является необязательным расширением, которое не будет доступным до тех пор, пока вы явно не включите его, выполнив оператор `from __future__ import with_statement` (см. главу 34).

История о двух `if`

Прежде чем мы погрузимся в детали конкретных операторов из табл. 10.1, я хочу начать обзор синтаксиса операторов Python, показав вам то, что вы *не* собираетесь вводить в коде Python. Это даст возможность сравнить синтаксис операторов Python с другими синтаксическими модулями, которые вы могли видеть в прошлом.

Взгляните на следующий оператор `if`, запрограммированный на C-подобном языке:

```
if (x > y) {
    x = 1;
    y = 2;
}
```

Оператор может относиться к C, C++, Java, JavaScript или похожему языку. А вот эквивалентный оператор в языке Python:

```
if x > y:
    x = 1
    y = 2
```

Первое, что может броситься в глаза — эквивалентный оператор Python не настолько загроможден, т.е. в нем меньше синтаксических компонентов. Так было задумано; одна из целей Python как языка написания сценариев заключается в том, чтобы облегчить жизнь программистов, требуя меньшего объема набора.

В частности, сравнив две синтаксические модели, вы заметите, что Python добавляет один элемент к смеси, а три элемента, присутствующие в C-подобном языке, в коде Python отсутствуют.

Что Python добавляет

Новым компонентом синтаксиса в Python является символ двоеточия (`:`). Все *составные операторы* Python, т.е. операторы с вложенными другими операторами, соблюдают общий шаблон. Шаблон состоит из строки заголовка, завершаемой двоеточием, и вложенного блока кода, обычно указываемого с отступом под строкой заголовка:

```
Строка заголовка:
    Вложенный блок операторов
```

Двоеточие обязательно, а его отсутствие является, пожалуй, наиболее распространенной ошибкой, допускаемой начинающими программистами на Python — безусловно, мне тысячи раз приходилось быть свидетелем такой ошибки в группах студентов, которых я обучал. На самом деле, если вы новичок в Python, вы почти наверняка вскоре забудете о символе двоеточия. В таком случае вы получите сообщение об ошибке, а большинство дружественных к Python редакторов позволят легко выявить такое недоразумение. Со временем включение двоеточия становится неосознанной привычкой (настолько сильной, что вы можете начать вводить двоеточия также и в коде на С-подобном языке, приводя к выдаче забавных сообщений об ошибках компилятором этого языка!).

Что Python устраняет

Несмотря на то что Python требует добавочного символа двоеточия, программистам на С-подобных языках приходится включать три элемента, которые обычно в Python не нужны.

Круглые скобки необязательны

Первый элемент — набор круглых скобок вокруг проверок в верхней части оператора:

```
if (x < y)
```

Круглые скобки здесь требуются синтаксисом многих С-подобных языков. Тем не менее, в Python они необязательны — мы просто опускаем круглые скобки, и оператор работает аналогичным образом:

```
if x < y
```

Говоря формально, из-за того, что любое выражение может быть помещено в круглые скобки, их наличие не повредит в таком коде Python, и они не трактуются как ошибка, когда присутствуют.

Но не поступайте так: вы будете без нужды изнашивать свою клавиатуру и заявлять всему миру о том, что являетесь программистом на С-подобном языке, все еще изучающим Python (я знаю, потому как сам был таким). “Образ действий Python” заключается в том, чтобы просто опускать круглые скобки в операторах такого вида.

Конец строки является концом оператора

Второй и более важный элемент синтаксиса, который вы не обнаружите в коде Python — точка с запятой. В Python вы не обязаны завершать операторы точками с запятой, как принято в С-подобных языках:

```
x = 1;
```

Общее правило в Python состоит в том, что конец строки автоматически завершает оператор, находящийся в этой строке. Другими словами, вы можете избавиться от точки с запятой, и оператор будет работать прежним образом:

```
x = 1
```

Как вы вскоре увидите, есть несколько способов обойти указанное правило (скажем, помещение кода внутрь структуры в квадратных скобках позволяет разнести ее на множество строк). Но в подавляющем большинстве кода вы будете записывать по одному оператору в строке, и никакие точки с запятой не потребуются.

Если вы тоскуете по тем дням, когда программировали на C (при условии, что такое вообще возможно), то можете продолжать использовать точки с запятой в конце каждого оператора — язык в состоянии смириться с их присутствием, поскольку точка с запятой служит также разделителем при комбинировании операторов.

Но не поступайте так тоже (серьезно!). Тем самым вы опять сообщаете миру о том, что являетесь программистом на C-подобном языке, который еще не полностью переключился на стиль написания кода Python. Стиль Python предусматривает отказ от точек с запятой. Судя по студентам в группах, некоторым программистам со стажем, похоже, нелегко отказаться от такой привычки. Но вы добьетесь своего; точки с запятой в роли завершения операторов — бесполезные помехи в Python.

Конец отступа является концом блока

Третий и последний элемент синтаксиса, устраненный в Python, отсутствие которого может выглядеть самым необычным для тех, кто недавно прекратил программировать на C-подобных языках, связан с тем, что вы не набираете в коде что-нибудь явное для синтаксической пометки начала и конца вложенного блока кода. Вам не нужно помещать вложенный блок внутрь `begin/end`, `then/endif` или фигурных скобок, как вы поступали бы в C-подобных языках:

```
if (x > y) {
    x = 1;
    y = 2;
}
```

В Python взамен мы согласованно смещаем все операторы в заданном одиночном вложенном блоке на одно и то же расстояние вправо, и для определения начала и конца блока Python применяет физические отступы операторов:

```
if x > y:
    x = 1
    y = 2
```

Под *отступом* здесь подразумевается пустое пространство слева от двух вложенных операторов. Python вовсе не заботит то, *как* сделан отступ (можно использовать либо пробелы, либо табуляции), или то, насколько отступ *большой* (допускается применять любое количество пробелов или табуляций). На самом деле отступ одного вложенного блока может совершенно отличаться от отступа другого. Синтаксическое правило лишь гласит о том, что все операторы заданного одиночного вложенного блока должны быть смещены на то же самое расстояние вправо. В противном случае возникает синтаксическая ошибка, а код не запустится до тех пор, пока вы не приведете отступы в согласованное состояние.

Для чего используется синтаксис с отступами?

Правило отступа может показаться на первый взгляд необычным программистам, привыкшим к C-подобным языкам, но это преднамеренная особенность Python и один из главных способов, которыми Python вынуждает программистов производить единообразный, систематический и читабельный код. Правило отступа по существу означает, что вы обязаны выравнивать свой код вертикально по столбцам в соответствии с его логической структурой. Совокупный эффект заключается в том, что код становится более согласованным и читабельным (в отличие от большинства кода на C-подобных языках).

Выражаясь более строго, выравнивание кода в соответствии с его логической структурой является значительной работой по содействию его читабельности и тем самым многократному использованию и удобству сопровождения, как вами, так и другими. В действительности, даже если вы не будете применять Python после прочтения книги, то все равно должны выработать привычку выравнивать свой код в целях читабельности в любом блочно-структурированном языке. Python акцентирует внимание на данной задаче, сделав выравнивание частью синтаксиса, но поступать так важно в любом языке программирования, что оказывает огромное влияние на полезность результирующего кода.

Ваш опыт может отличаться, но когда я все еще занимался разработкой на постоянной основе, мне главным образом платили за работу над крупными старыми программами C++, которые создавались многими программистами в течение нескольких лет. Почти у каждого программиста был свой стиль отступов в коде. Например, меня часто просили изменять цикл `while` в коде C++, который начинался примерно так:

```
while (x > 0) {
```

До того, как мы углубимся в отступы, следует отметить, что есть три или четыре способа, которыми программисты размещают фигурные скобки в C-подобном языке, и в организациях часто ведутся горячие споры и составляются руководства по стандартам, регламентирующие допустимые варианты (что выглядит более чем просто отклонением от задач, решаемых с помощью программирования). Как бы то ни было, вот вам сценарий, который я часто встречал в коде C++. Первый человек, работающий над кодом, использовал для тела цикла отступ в четыре пробела:

```
while (x > 0) {
    --------;
    --------;
```

Со временем этот человек перешел на руководящую должность, а его заменил другой человек, которому нравилось делать еще большие отступы вправо:

```
while (x > 0) {
    --------;
    --------;
        --------;
        --------;
```

Позже любитель широких отступов воспользовался возможностью и сменил работу (положив конец царствованию кодового террора в своем лице...), а его место занял человек, который предпочитал делать поменьше отступов:

```
while (x > 0) {
    --------;
    --------;
        --------;
        --------;
--------;
--------;
}
```

И так далее. В конце концов, блок завершается закрывающей фигурной скобкой (`}`), которая, конечно же, делает его “блочно-структурированным кодом” (здесь присутствует изрядная доля сарказма). Нет: в любом блочно-структурированном языке, Python или еще каком, если вложенные блоки не имеют согласованных отступов, тогда их становится очень трудно истолковывать, изменять или повторно использовать, пото-

му что код больше визуально не отражает свой логический смысл. *Читабельность имеет значение*, и отступы являются важной составной частью читабельности.

Ниже приведен пример, на котором вы могли обжечься в прошлом, если много программировали на каком-то C-подобном языке. Взгляните на следующий оператор в C:

```
if (x)
    if (y)
        оператор1;
else
    оператор2;
```

К какому оператору `if` здесь относится `else`? Как ни удивительно, но в C часть `else` относится к вложенному оператору `if` (`if (y)`), несмотря на то, что визуально выглядит связанной с внешним оператором `if (x)`. Это классическая ловушка в языке C; она может привести к тому, что читатель совершенно неправильно интерпретирует код и некорректно изменяет его способами, которые иногда обнаруживаются лишь после того, как марсоход врезался в огромную скалу!

В Python подобное произойти не может — из-за того, что отступы существенны, код работает именно так, как выглядит. Вот эквивалентный оператор Python:

```
if x:
    if y:
        оператор1
else:
    оператор2
```

В приведенном примере часть `else` логически связана с оператором `if`, с которым выровнена по вертикали (внешний `if x`). В известной степени Python является языком WYSIWYG (what you see is what you get — что видишь, то и получаешь), поскольку внешний вид кода определяет путь его выполнения независимо от того, кто его писал.

Если сказанного по-прежнему не хватает, чтобы должным образом оценить преимущества синтаксиса Python, то приведу еще один эпизод. В начале своей карьеры я работал в успешной компании, которая занималась разработкой системного программного обеспечения на языке C, где согласованные отступы не требовались. Даже в таких условиях, когда в конце дня мы сохраняли свой код в системе управления версиями исходного кода, в компании автоматически запускался сценарий, который анализировал отступы, применяемые в коде. Если сценарий замечал, что мы использовали отступы в коде несогласованно, то на следующее утро мы получали по электронной почте соответствующее сообщение — равно как и наши менеджеры!

Дело в том, что даже когда язык этого не требует, хорошим программистам известно, что согласованное применение отступов оказывает сильное влияние на читабельность и качество кода. Тот факт, что отступы в Python продвинуты до уровня синтаксиса, рассматривается большинством как особенность языка.

Также имейте в виду, что почти каждый дружественный к программистам текстовый редактор обладает встроенной поддержкой синтаксической модели Python. Скажем, в графическом пользовательском интерфейсе IDLE к строкам кода автоматически добавляется отступ при наборе вложенного блока; нажатие клавиши забоя возвращает обратно на один уровень отступов, и то, насколько далеко вправо IDLE смещает операторы во вложенном блоке, можно настраивать. Универсального стандарта не существует: общепринятыми являются четыре пробела или одна табуляция, но обычно вы сами решаете, каким образом и насколько отступать (если только не работаете в компании, где отступы регламентируются политикой и внутренними стан-

дартами). Делайте больший отступ вправо для следующего вложенного блока и меньший отступ для закрытия предыдущего блока.

В качестве эмпирического правила запомните, что вероятно не стоит смешивать табуляции и пробелы в том же самом блоке в Python, если только не делать это согласованно; в отдельно взятом блоке используйте табуляции или пробелы, но не то и другое (на самом деле, как будет показано в главе 12, теперь Python 3.X сообщает об ошибке при несогласованном применении табуляций и пробелов). Более того, смешивать табуляции и пробелы в отступах, видимо, не следует в *любом* структурированном языке — такой код может вызвать крупные проблемы с читабельностью, если текстовый редактор у следующего программиста настроен на отображение табуляций не так, как у вас. С-подобные языки зачастую разрешают программистам обходить данное правило, но программисты не должны поступать так: результатом может оказаться изрядно запутанная смесь.

Независимо от языка, на котором пишется код, вы должны придерживаться согласованных отступов для обеспечения читабельности. Фактически, если вас не приучили делать это раньше, то ваши учителя оказали вам медвежью услугу. Большинство программистов (особенно те, кому приходится читать код, написанный другими) считает ценным качеством то, что Python продвигает правило, касающееся отступов, до уровня синтаксиса. Кроме того, инструментам, которые должны выводить код Python, не составляет труда генерировать табуляции вместо фигурных скобок. В целом, если вы делаете то, что в любом случае должны были бы делать в С-подобном языке, но избавляетесь от фигурных скобок, тогда ваш код будет удовлетворять правилам синтаксиса Python.

Несколько специальных случаев

Как упоминалось ранее, в синтаксической модели Python:

- конец строки завершает оператор в этой строке (безо всяких точек с запятой);
- вложенные операторы объединяются в блок и ассоциируются согласно их физическим отступам (без фигурных скобок).

Указанные правила охватывают почти весь код Python, который вам доведется писать или встречать в реальности. Однако Python также предлагает ряд специализированных правил, которые делают возможной настройку как операторов, так и вложенных блоков операторов. Они не обязательны и должны использоваться умеренно, но программисты находят их полезными на практике.

Специальные правила для операторов

Хотя операторы обычно располагают по одному в строке, в Python разрешено помещать несколько операторов в одну строку, разделяя их точками с запятой:

```
a = 1; b = 2; print(a + b)    # Три оператора в одной строке
```

Это единственное место в Python, где требуются точки с запятой: в качестве *разделителей между операторами*. Тем не менее, такое применение допускается, только если объединяемые подобным образом операторы сами не являются составными. Другими словами, вы можете выстраивать в цепочку лишь простые операторы вроде присваиваний, вызовов `print` и обращений к функциям. Составные операторы, такие как проверки `if` и циклы `while`, по-прежнему должны находиться в собственных строках (иначе вы могли бы втиснуть всю программу в одну строку, что вряд ли добавило бы вам популярности среди коллег по работе!).

Другое специальное правило для операторов по существу является противоположностью: вы можете разнести одиночный оператор на *множество строк*. Для этого понадобится лишь поместить часть оператора внутрь пары скобок — круглых (`()`), квадратных (`[]`) или фигурных (`{}`). Любой код, заключенные в такие скобки, может занимать несколько строк: оператор не закончится до тех пор, пока Python не достигнет закрывающей скобки из пары. Например, вот как записать списковый литерал в нескольких строках:

```
mylist = [1111,
          2222,
          3333]
```

Поскольку код заключен в пару квадратных скобок, Python просто переходит на следующую строку до тех пор, пока не встретит закрывающую квадратную скобку. Фигурные скобки, окружающие словари (а также литералы множеств и включения словарей и множеств в Python 3.X/2.7), позволяют им распространяться на несколько строк, а круглые скобки делают это для кортежей, вызовов функций и выражений. Отступы в строках продолжения роли не играют, хотя здравый смысл подсказывает, что строки должны каким-то образом выравниваться ради читабельности.

Круглые скобки представляют собой универсальное средство — из-за того, что в них можно помещать любое выражение, вставка открывающей круглой скобки дает возможность продолжить оператор в следующей строке:

```
X = (A + B +
     C + D)
```

Кстати, такая методика работает также и с составными операторами. Везде, где нужно записать крупное выражение, просто поместите его в круглые скобки, чтобы перенести на следующую строку:

```
if (A == 1 and
    B == 2 and
    C == 3):
      print('spam' * 3)
```

Более старое правило позволяет продолжать оператор в следующей строке, если предыдущая строка заканчивается на обратную косую черту:

```
X = A + B + \
     C + D    # Подверженная ошибкам более старая альтернатива
```

Однако такая альтернативная методика вышла из употребления и в наши дни не одобряется, потому что замечать и сохранять обратные косые черты нелегко. Она также довольно хрупкая и подвержена ошибкам. Дело в том, что после обратной косой черты не должно быть пробелов, а случайный пропуск обратной косой черты может приводить к неожиданным эффектам, если следующая строка будет ошибочно воспринята как новый оператор. (В показанном выше примере `C + D` — сам по себе допустимый оператор, если он не имеет отступа.) Такое правило также является еще одним отголоском языка C, где оно обычно используется в макросах `#define`; если вы находитесь в мире Python, то и поступайте, как принято в Python, а не в C.

Специальные правила для блоков

Как упоминалось ранее, операторы во вложенном блоке кода обычно ассоциируются по их отступам на одно и то же расстояние вправо. В качестве одного специального случая здесь тело составного оператора может взамен находиться в той же самой строке, что и строка заголовка оператора Python, после двоеточия:

```
if x > y: print(x)
```

В результате у нас появляется возможность записывать однострочные операторы `if`, однострочные циклы `while` и `for` и т.д. Тем не менее, это будет работать, только если тело составного оператора само не содержит каких-либо составных операторов. То есть после двоеточия разрешено указывать лишь простые операторы — присваивания, вызовы `print`, обращения к функциям и т.п. Более крупные операторы по-прежнему должны находиться в собственных строках. Дополнительные части составных операторов (такие как часть `else` оператора `if`, который мы рассмотрим в следующем разделе) также обязаны располагаться в отдельных строках. Тела составных операторов могут состоять из множества простых операторов, разделенных точками с запятой, но обычно такой подход не одобряется.

В общем, хотя временами это необязательно, если вы будете размещать все операторы в отдельных строках и всегда делать отступы для вложенных блоков, то ваш код станет легче читать и изменять в будущем. Более того, некоторые инструменты для профилирования и покрытия кода могут быть не в состоянии проводить различия между множеством операторов, втиснутых в одну строку, или заголовком и телом однострочного составного оператора. Практически всегда в ваших интересах поддерживать в Python простоту. Вы можете применять специальные правила для написания кода Python, который трудно читать, но это требует немало работы, и у вас наверняка найдется, куда лучше потратить свое время.

Однако чтобы увидеть в действии простое и распространенное исключение из указанных правил (использование однострочного оператора `if` для выхода из цикла с помощью `break`), а также представить дополнительный синтаксис Python, в следующем разделе будет написан реальный код.

Короткий пример: интерактивные циклы

Мы увидим все описанные правила синтаксиса в действии, когда начнем тур по специфическим составным операторам Python в последующих нескольких главах, но повсюду в языке Python они работают одинаково. Давайте пока рассмотрим короткий, но реалистичный пример, который продемонстрирует практический способ совместного применения синтаксиса операторов и вложения операторов, а также попутно ознакомит с несколькими операторами.

Простой интерактивный пример

Предположим, что вас попросили написать на Python программу, которая взаимодействует с пользователем в окне консоли. Возможно, необходимо принимать ввод для отправки в базу данных или читать числа с целью использования в вычислении. Независимо от предназначения, вам придется написать код цикла, который читает одну или большее количество строк, набираемых пользователем на клавиатуре, и выводит для каждой результат. Другими словами, вы должны написать программу с классическим циклом чтение/оценка/вывод.

Типичный шаблонный код для такого цикла взаимодействия может выглядеть в Python следующим образом:

```
while True:
    reply = input('Enter text:')
    if reply == 'stop': break
    print(reply.upper())
```

В коде применяется несколько новых идей и ряд тех, что вы уже видели ранее.

- В коде задействован цикл `while` языка Python — наиболее универсальный оператор цикла. Позже мы детально исследуем оператор `while`, а пока достаточно знать, что он состоит из слова `while`, за которым следует выражение, дающее истинный или ложный результат, а за ним вложенный блок кода, повторяющийся до тех пор, пока выражение остается истинным (слово `True` здесь считается всегда истинным).
- Встроенная функция `input`, которую мы встречали ранее в книге, используется для универсального консольного ввода — она выводит необязательный аргумент в качестве приглашения и возвращает то, что пользователь набрал, в виде строки. Согласно приведенной далее врезке "На заметку!" в Python 2.X взамен применяется `raw_input`.
- В коде также присутствует однострочный оператор `if`, который использует специальное правило для вложенных блоков: тело оператора `if` находится в строке заголовка после двоеточия вместо того, чтобы располагаться с отступом в строке ниже. В любом случае оператор будет работать одинаково, но так мы экономим одну строку.
- Наконец, оператор `break` языка Python применяется для немедленного выхода из цикла — происходит просто переход из цикла и программа продолжает выполнение сразу после оператора цикла. Без такого оператора выхода цикл `while` выполнялся бы бесконечно, потому что его проверка всегда истинна.

В сущности, приведенная комбинация операторов означает "читать строки, введенные пользователем, и выводить их в верхнем регистре до тех пор, пока пользователь не введет слово `stop`". Существуют и другие способы написания такого цикла, но использованная здесь форма очень часто встречается в коде Python.

Обратите внимание, что все три строки, вложенные под строку заголовка `while`, смещены на одно и то же расстояние вправо — поскольку операторы подобным образом выровнены вертикально по столбцам, они являются блоком кода, который ассоциирован с проверкой `while` и повторяется. Для завершения блока с телом цикла будет достаточно либо конца файла исходного кода, либо оператора с меньшим отступом.

Когда код запускается, интерактивно или как файл сценария, вот какое взаимодействие мы получаем (весь код примера находится в файле `interact.py` из пакета примеров для этой книги):

```
Enter text:spam
SPAM
Enter text:42
42
Enter text:stop
```

Примечание, касающееся нестыковки версий. В примере написан код для Python 3.X. Если вы работаете в Python 2.X, то код работает так же, но во всех примерах текущей главы вместо `input` придется применять `raw_input`, и можно опустить внешние круглые скобки в операторах `print` (хотя никакого вреда они не наносят). В действительности, если вы исследуете файл `interact.py` из пакета примеров, то увидите, что в нем это делается автоматически — для поддержки совместимости с Python 2.X имя `input` переустанавливается, если старшим номером версии функционирующего Python является 2 (вызов `input` приводит к выполнению `raw_input`):

```
import sys
if sys.version[0] == '2': input = raw_input  # Совместимость
                                             #   с Python 2.X
```

В Python 3.X имя `raw_input` было изменено на `input`, а `print` представляет собой встроенную функцию, а не оператор (более подробно `print` обсуждается в следующей главе). В Python 2.X также имеется оператор `input`, но он пытается выполнить входную строку, как если бы она было кодом Python, что вероятно не будет работать в данном контексте; в Python 3.X того же эффекта можно достичь посредством `eval(input())`.

Выполнение математических действий над пользовательским вводом

Наш сценарий работает, но теперь предположим, что вместо преобразования текстовой строки в верхний регистр мы хотим выполнить какие-то математические действия над числовым вводом — скажем, возвести его в квадрат, возможно в ошибочной попытке программы ввода возраста подразнить своих пользователей. Для получения желаемого результата мы могли бы использовать код вроде показанного ниже:

```
>>> reply = '20'
>>> reply ** 2
...текст сообщения об ошибке не показан...
TypeError: unsupported operand type(s) for ** or pow(): 'str' and 'int'
Ошибка типа: неподдерживаемые типы операндов для ** или pow(): str и int
```

Тем не менее, прием в нашем сценарии работать не будет, потому что (как обсуждалось в предыдущей части книги) Python не преобразует типы в выражениях, если только все они не являются числовыми, а ввод от пользователя всегда возвращается сценарию в виде *строки*. Мы не сможем возвести строку цифр в степень до тех пор, пока вручную не преобразуем ее в целое число:

```
>>> int(reply) ** 2
400
```

Вооружившись этой информацией, мы можем переписать наш цикл для выполнения необходимого математического действия. Поместите следующий код в файл для его тестирования:

```
while True:
    reply = input('Enter text:')
    if reply == 'stop': break
    print(int(reply) ** 2)
print('Bye')
```

Как и ранее, в сценарии применяется однострочный оператор `if` для выхода из цикла при вводе `stop`, но также осуществляется преобразование ввода для выполнения требуемого математического действия. В конце еще добавляется прощальное сообщение. Поскольку оператор `print` в последней строке не имеет такого же отступа, как у вложенного блока кода, он не считается частью тела цикла и будет выполнен только раз после выхода из цикла:

```
Enter text:2
4
Enter text:40
1600
Enter text:stop
Bye
```

Замечание по использованию. С этого момента я буду предполагать, что код хранится в файле сценария и запускается из него через командную строку, пункт меню IDLE или любым другим способом запуска файлов, которые обсуждались в главе 3. В пакете примеров файл называется `interact.py`. Однако если вы вводите этот код интерактивно, тогда удостоверьтесь в том, что включили пустую строку (т.е. два раза нажали клавишу <Enter>) перед финальным оператором `print`, чтобы завершить цикл. Это также подразумевает невозможность вырезания и вставки кода в интерактивную подсказку: добавочная пустая строка требуется в интерактивном режиме, но не в файле сценария. Хотя в наборе последнего оператора `print` в интерактивном режиме мало смысла — вы будете вводить его после взаимодействия с циклом!

Обработка ошибок путем проверки ввода

Пока все хорошо, но обратите внимание, что происходит в случае ввода недопустимой строки:

```
Enter text:xxx
...текст сообщения об ошибке не показан...
ValueError: invalid literal for int() with base 10: 'xxx'
Ошибка значения: недопустимый литерал для int() с основанием 10: xxx
```

Столкнувшись с ошибкой, встроенная функция `int` генерирует исключение. Если нужно, чтобы сценарий был надежным, тогда можно заранее проверить содержимое строки с помощью метода `isdigit` строкового объекта:

```
>>> S = '123'
>>> T = 'xxx'
>>> S.isdigit(), T.isdigit()
(True, False)
```

Это также дает повод к дальнейшему вложению операторов в примере. В приведенной далее обновленной версии нашего интерактивного сценария с применением полнофункционального оператора `if` исключение при ошибках обходится:

```
while True:
    reply = input('Enter text:')
    if reply == 'stop':
        break
    elif not reply.isdigit():
        print('Bad!' * 8)
    else:
        print(int(reply) ** 2)
print('Bye')
```

Оператор `if` подробно рассматривается в главе 12, но сейчас достаточно знать, что он представляет собой довольно легковесный инструмент для кодирования логики в сценариях. В своей полной форме оператор состоит из слова `if`, за которым следует проверка и связанный блок кода, одна или более необязательных проверок `elif` (“else if”) и блоков кода, а также необязательная часть `else` с ассоциированным блоком кода в качестве принимаемого по умолчанию. Python выполняет блок кода, связанный с первой проверкой, которая дает истину, работая сверху вниз, или часть `else`, если все проверки оказались ложными.

Части `if`, `elif` и `else` в предыдущем примере относятся к одному и тому же оператору, потому что все они выровнены по вертикали (т.е. используют одинако-

вый уровень отступа). Оператор if простирается от слова if до начала оператора print в последней строке сценария. В свою очередь весь блок if является частью цикла while, поскольку он располагается с отступом под строкой заголовка цикла. Подобного рода вложение операторов станет выглядеть естественным, как только вы освоитесь с ним.

После запуска нового сценария его код перехватывает ошибки до того, как они произойдут, и выводит сообщение, прежде чем продолжить работу (которое вероятно имеет смысл улучшить в более позднем выпуске), но ввод stop по-прежнему приводит к завершению, а допустимые числа возводятся в квадрат:

```
Enter text:5
25
Enter text:xyz
Bad!Bad!Bad!Bad!Bad!Bad!Bad!Bad!
Enter text:10
100
Enter text:stop
Bye
```

Обработка ошибок с помощью оператора try

Предыдущее решение работает, но позже в книге вы увидите, что самый универсальный способ обработки ошибок в Python предусматривает перехват и полное восстановление после них с применением оператора try языка Python. Мы будем исследовать этот оператор в части VII книги, а сейчас отметим, что использование try здесь может в итоге дать код, который многие сочтут более простым, чем предыдущая версия:

```
while True:
    reply = input('Enter text:')
    if reply == 'stop': break
    try:
        num = int(reply)
    except:
        print('Bad!' * 8)
    else:
        print(num ** 2)
print('Bye')
```

Новая версия кода работает в точности как предыдущая, но явную проверку на предмет ошибки мы заменили кодом, который предполагает успешное выполнение преобразования, и поместили его внутрь обработчика исключений для случаев, когда это не так. Другими словами, вместо обнаружения ошибки мы просто реагируем, когда она возникает.

Оператор try является еще одним составным оператором и следует такому же шаблону, как if и while. Он состоит из слова try, за которым находится основной блок кода (действие, подлежащее выполнению), затем часть except с кодом обработчика исключений и часть else, выполняемая в случае, если никакие исключения в части try не генерировались. Python сначала выполняет часть try, после чего либо часть except (если исключение произошло), либо часть else (если исключений не было).

С точки зрения вложения операторов, поскольку слова try, except и else имеют отступы на том же самом уровне, они считаются частью одного оператора try.

Обратите внимание, что часть `else` здесь ассоциирована с `try`, а не с `if`. Как вы увидите, в Python часть `else` может появляться в операторах `if`, но также может присутствовать в операторах `try` и циклах — ее отступ сообщает, к какому оператору она относится. В данном случае оператор `try` начинается со слова `try` и охватывает код, расположенный с отступом под словом `else`, потому что `else` имеет такой же отступ, как у `try`. Оператор `if` в этом коде однострочный и заканчивается после `break`.

Поддержка чисел с плавающей точкой

Позже в книге мы еще вернемся к оператору `try`. Пока достаточно знать, что поскольку оператор `try` можно применять для перехвата любой ошибки, он сокращает объем кода проверки на предмет ошибок, который приходится писать, и представляет собой очень универсальный подход к работе с необычными случаями. Скажем, при наличии уверенности в том, что `print` не потерпит неудачу, тогда пример мог бы стать еще короче:

```
while True:
    reply = input('Enter text:')
    if reply == 'stop': break
    try:
        print(int(reply) ** 2)
    except:
        print('Bad!' * 8)
print('Bye')
```

И если мы хотим поддерживать ввод чисел с плавающей точкой вместо только целых, например, то использовать `try` было бы гораздо легче, чем вручную проверять на предмет ошибок — мы могли бы просто вызвать функцию `float` и перехватить ее исключения:

```
while True:
    reply = input('Enter text:')
    if reply == 'stop': break
    try:
        print(float(reply) ** 2)
    except:
        print('Bad!' * 8)
print('Bye')
```

На сегодняшний день функции вроде `isfloat` для строк не предусмотрено, поэтому такой подход на основе исключений избавляет нас от необходимости анализировать весь возможный синтаксис чисел с плавающей точкой посредством явной проверки на предмет ошибок. Когда код написан таким способом, мы можем вводить более разнообразные числа, но выявление ошибок и выход по-прежнему работают, как и ранее:

```
Enter text:50
2500.0
Enter text:40.5
1640.25
Enter text:1.23E-100
1.5129e-200
Enter text:spam
Bad!Bad!Bad!Bad!Bad!Bad!Bad!Bad!
Enter text:stop
Bye
```

Здесь на месте `float` также работал бы вызов функции `eval` языка Python, который мы применяли в главах 5 и 9 для преобразования данных в строках и файлах. Тогда появилась бы возможность вводить произвольные выражения (`2 ** 100` было бы допустимым, хотя и необычным вводом, особенно с учетом того, что программа обрабатывает возраст!). Это мощная концепция, которая подвержена тем же самым проблемам с безопасностью, о которых шла речь в предшествующих главах. Если вы не можете доверять источнику строки с кодом, тогда используйте более ограничивающие инструменты преобразования наподобие `int` и `float`.

Функция `exec` языка Python, применяемая в главе 3 для выполнения кода из файла, похожа на `eval` (но предполагает, что строка является оператором, а не выражением, и не возвращает результат), а вызов `compile` предварительно компилирует часто используемые строки кода в объекты байт-кода ради достижения высокой скорости. Дополнительные сведения можно получить, запустив `help` для любого из указанных средств; как упоминалось ранее, `exec` представляет собой оператор в Python 2.X, но функцию в Python 3.X, поэтому в Python 2.X ищите ее описание в руководстве. Мы также будем применять `exec` в главе 25 для импортирования модулей, используя строку с именем (пример более динамических ролей данного инструмента).

Вложение кода на три уровня в глубину

Давайте рассмотрим последнее изменение кода. При необходимости вложение можно продолжить; скажем, мы могли бы расширить приведенный ранее сценарий, обрабатывающий только целые числа, для перехода к одной из набора альтернатив на основе относительной величины допустимого ввода:

```
while True:
    reply = input('Enter text:')
    if reply == 'stop':
        break
    elif not reply.isdigit():
        print('Bad!' * 8)
    else:
        num = int(reply)
        if num < 20:
            print('low')
        else:
            print(num ** 2)
print('Bye')
```

В последней версии сценария добавлен оператор `if`, вложенный в часть `else` другого оператора `if`, который в свою очередь вложен в цикл `while`. Когда код является условным или повторяется как здесь, мы просто смещаем его дальше вправо. Совокупный эффект похож на предшествующие версии, но теперь для чисел, меньших 20, будет выводиться слово `low`:

```
Enter text:19
low
Enter text:20
400
Enter text:spam
Bad!Bad!Bad!Bad!Bad!Bad!Bad!Bad!
Enter text:stop
Bye
```

Резюме

На этом краткий обзор синтаксиса операторов Python завершен. В главе были представлены общие правила для написания операторов и блоков кода. Вы узнали, что в Python мы обычно записываем по одному оператору в строке и смещаем все операторы во вложенном блоке на одно и то же расстояние вправо (отступ — часть синтаксиса Python). Кроме того, мы также взглянули на ряд исключений из этих правил, в том числе строки продолжения и однострочные проверки и циклы. Наконец, мы воплотили все рассмотренные идеи на практике в сценарии с интерактивным циклом, где продемонстрировали несколько операторов и показали синтаксис операторов в действии.

В следующей главе мы начнем более тщательно исследовать каждый базовый процедурный оператор Python. Как вы увидите, все операторы соблюдают те же самые общие правила, которые были описаны в настоящей главе.

Проверьте свои знания: контрольные вопросы

1. Какие три элемента синтаксиса обязательны в C-подобном языке, но опущены в Python?
2. Как обычно завершается оператор в Python?
3. Как операторы во вложенном блоке обычно ассоциируются в Python?
4. Как можно было бы разместить одиночный оператор в нескольких строках?
5. Как можно было бы записать составной оператор в одной строке?
6. Есть ли веские причины набирать точку с запятой в конце оператора в Python?
7. Для чего предназначен оператор `try`?
8. Какую ошибку чаще всего допускают новички в Python?

Проверьте свои знания: ответы

1. C-подобные языки требуют наличия круглых скобок вокруг выражений проверки в некоторых операторах, точки с запятой в конце каждого оператора и фигурных скобок вокруг вложенного блока кода.
2. Конец строки завершает оператор, находящийся в этой строке. Если в одной строке присутствует несколько операторов, то их можно завершать с помощью точек с запятой; подобным же образом, если оператор охватывает множество строк, тогда его придется завершать путем помещения внутрь пары квадратных скобок.
3. Все операторы во вложенном блоке имеют отступы с одинаковым количеством табуляций или пробелов.
4. Разместить одиночный оператор в нескольких строках можно, поместив его часть в круглые, квадратные или фигурные скобки; оператор заканчивается, когда Python встречает строку, в которой содержится закрывающая скобка из пары.
5. Тело составного оператора может быть перемещено в строку заголовка после двоеточия, но только если тело состоит из несоставных операторов.
6. Лишь тогда, когда необходимо втиснуть несколько операторов в одну строку кода. И то это работает, только если операторы являются несоставными, к тому же не одобряется, поскольку может давать в итоге код, который трудно читать.
7. Оператор `try` применяется для перехвата и восстановления после исключений (ошибок) в сценарии Python. Он обычно выступает в качестве альтернативы ручной проверки на предмет ошибок в коде.
8. Самая распространенная ошибка новичков связана с тем, что они забывают набрать двоеточие в конце строки заголовка составного оператора. Если вы начали изучать Python и пока ее не допустили, то вероятно это скоро произойдет!