16

Области видимости и аргументы

В главе 15 были описаны основы определения и вызова функций. Как мы видели, базовая модель функций в языке Python достаточно проста в использовании. В этой главе будут представлены подробности, лежащие в основе *областей видимости* – мест, где определяются переменные и где выполняется их поиск, – и механизма *передачи аргументов* – способа передачи объектов в функции.

Правила видимости

Теперь, когда вы готовы приступить к созданию своих собственных функций, нам необходимо более формально определить, что означают имена в языке Python. Всякий раз, когда в программе используется некоторое имя, интерпретатор создает, изменяет или отыскивает это имя в *пространстве имен* – в области, где находятся имена. Когда мы говорим о поиске значения имени применительно к программному коду, под термином *область видимости* подразумевается пространство имен: т. е. место в программном коде, где имени было присвоено значение, определяет область видимости этого имени для программного кода.

Практически все, что имеет отношение к именам, включая классификацию областей видимости, в языке Python связано с операциями присваивания. Как мы уже видели, имена появляются в тот момент, когда им впервые присваиваются некоторые значения, и прежде чем имена смогут быть использованы, им необходимо присвоить значения. Поскольку имена не объявляются заранее, интерпретатор Python по местоположению операции присваивания связывает имя с конкретным пространством имен. Другими словами, место, где выполняется при-

сваивание, определяет пространство имен, в котором будет находиться имя, а, следовательно, и область его видимости.

Помимо упаковки программного кода функции привносят в программы еще один слой пространства имен – по умолчанию все имена, значения которым присваиваются внутри функции, ассоциируются с пространством имен этой функции и никак иначе. Это означает, что:

- Имена, определяемые внутри инструкции `def`, видны только программному коду внутри инструкции `def`. К этим именам нельзя обратиться за пределами функции.
- Имена, определяемые внутри инструкции `def`, не вступают в конфликт с именами, находящимися за пределами инструкции `def`, даже если и там и там присутствуют одинаковые имена. Имя `X`, которому присвоено значение за пределами данной инструкции `def` (например, в другой инструкции `def` или на верхнем уровне модуля) полностью отлично от имени `X`, которому присвоено значение внутри инструкции `def`.

В любом случае область видимости переменной (где она может использоваться) всегда определяется местом, где ей было присвоено значение, и никакого отношения не имеет к месту, откуда была вызвана функция. Если присваивание переменной выполняется внутри инструкции `def`, переменная является *локальной* для этой функции; если присваивание производится за пределами инструкции `def`, она является *глобальной* для всего файла. Мы называем это *лексической областью видимости*, потому что области видимости переменных целиком определяются местоположением этих переменных в исходных текстах программы, а не местом, откуда вызываются функции.

Например, в следующем файле модуля инструкция присваивания `X = 99` создает глобальную переменную с именем `X` (она видима из любого места в файле), а инструкция `X = 88` создает локальную переменную `X` (она видима только внутри инструкции `def`):

```
X = 99

def func():
    X = 88
```

Даже при том, что обе переменные имеют имя `X`, области видимости делают их различными. Таким образом, области видимости функций позволяют избежать конфликтов имен в программах и превращают функции в самостоятельные элементы программ.

Основы видимости имен в языке Python

До того как мы начали писать функции, весь программный код размещался на верхнем уровне модуля (например, он не был вложен в инструкции `def`), поэтому все имена, которые мы использовали, либо нахо-

дились на верхнем уровне в модуле, либо относились к предопределенным именам языка Python (например, `open`).[1] Функции образуют вложенные пространства имен (области видимости), которые ограничивают доступ к используемым в них именам, благодаря чему имена внутри функций не вступают в конфликт с именами за их пределами (внутри модуля или внутри других функций). Повторю еще раз, функции образуют *локальную область видимости*, а модули – глобальную. Эти две области взаимосвязаны между собой следующим образом:

- **Объемлющий модуль – это глобальная область видимости.** Каждый модуль – это глобальная область видимости, т. е. пространство имен, в котором создаются переменные на верхнем уровне в файле модуля. Глобальные переменные для внешнего мира становятся атрибутами объекта модуля, но внутри модуля могут использоваться как простые переменные.
- **Глобальная область видимости охватывает единственный файл.** Не надо заблуждаться насчет слова «глобальный» – имена на верхнем уровне файла являются глобальными только для программного кода в этом файле. На самом деле в языке Python не существует такого понятия, как всеобъемлющая глобальная для всех файлов область видимости. Имена всегда относятся к какому-нибудь модулю и всегда необходимо явно импортировать модуль, чтобы иметь возможность использовать имена, определяемые в нем. Когда вы слышите слово «глобальный», подразумевайте «модуль».
- **Каждый вызов функции создает новую локальную область видимости.** Всякий раз, когда вызывается функция, создается новая локальная область видимости – т. е. пространство имен, в котором находятся имена, определяемые внутри функции. Каждую инструкцию `def` (и выражение `lambda`) можно представить себе как определение новой локальной области видимости. Но так как язык Python позволяет функциям вызывать самих себя в цикле (этот прием известен как *рекурсия*), локальная область видимости с технической точки зрения соответствует вызову функции – другими словами, каждый вызов создает новое локальное пространство имен. Рекурсию удобно использовать, когда выполняется обработка данных, структура которых заранее неизвестна.
- **Операция присваивания создает локальные имена, если они не были объявлены глобальными.** По умолчанию все имена, которым присваиваются значения внутри функции, помещаются в локальную область видимости (пространство имен, ассоциированное с вызовом функции). Если необходимо присвоить значение имени верх-

[1] Программный код, который вводится в интерактивной оболочке, в действительности находится на уровне модуля `__main__`, поэтому имена, создаваемые в интерактивной оболочке, также находятся внутри модуля и следуют обычным правилам видимости. Подробнее о модулях будет рассказываться в пятой части книги.

него уровня в модуле, который вмещает функцию, это имя необходимо объявить внутри функции глобальным с помощью инструкции `global`.

- **Все остальные имена являются локальными в объемлющей области видимости, глобальными или встроенными.** Предполагается, что имена, которым не присваивались значения внутри определения функции, находятся в объемлющей локальной области видимости (внутри объемлющей инструкции `def`), глобальной (в пространстве имен модуля) или встроенной (предопределенные имена в модуле `__builtin__`).

Обратите внимание, что любые операции присваивания, выполняемые внутри функции, классифицируют имена как локальные: инструкция `=`, инструкция `import`, инструкция `def`, передача аргументов и т. д. Следует также заметить, что операции непосредственного изменения объектов не рассматривают имена как локальные – это свойственно только операциям присваивания. Например, если имени `L` присвоен список, определенный на верхнем уровне в модуле, то такая инструкция, как `L.append(X)`, внутри функции не будет классифицировать имя `L` как локальное, тогда как инструкция `L = X` будет. В первом случае список `L` будет найден в глобальной области видимости и инструкция изменит глобальный список.

Разрешение имен: правило LEGB

Если предыдущий раздел показался вам запутанным, спешу успокоить – в действительности все сводится к трем простым правилам. Для инструкции `def`:

- Поиск имен ведется самое большее в четырех областях видимости: локальной, затем в объемлющей функции (если таковая имеется), затем в глобальной и, наконец, во встроенной.
- По умолчанию операция присваивания создает локальные имена.
- Глобальные объявления отображают имена на область видимости вмещающего модуля.

Другими словами, все имена, которым присваиваются значения внутри инструкции `def` (или внутри выражения `lambda`, с которым мы познакомимся позже), по умолчанию являются локальными; функции могут использовать имена в лексически (т. е., физически) объемлющих функциях и в глобальной области видимости, но чтобы иметь возможность изменять их, они должны быть объявлены глобальными. Схема разрешения имен в языке Python иногда называется *правилом LEGB*, название которого состоит из первых букв названий областей видимости:

- Когда внутри функции выполняется обращение к неизвестному имени, интерпретатор пытается отыскать его в четырех областях видимости – в локальной (local, *L*), затем в локальной области любой объемлющей инструкции `def` (enclosing, *E*) или в выражении

`lambda`, затем в глобальной (global, *G*) и, наконец, во встроенной (built-in, *B*). Поиск завершается, как только будет найдено первое подходящее имя.

- Когда внутри функции выполняется операция присваивания (а не обращение к имени внутри выражения), интерпретатор всегда создает или изменяет имя в локальной области видимости, если в этой функции оно не было объявлено глобальным.
- Когда выполняется присваивание имени за пределами функции (т. е. на уровне модуля или в интерактивной оболочке), локальная область видимости совпадает с глобальной – с пространством имен модуля.

На рис. 16.1 показаны четыре области видимости в языке Python. Примечательно, что второй области видимости *E* – в области видимости объемлющей инструкции `def` или выражения `lambda`, с технической точки зрения может находиться несколько вложенных друг в друга областей. Но они появляются, только когда имеются вложенные друг в друга функции.[1]

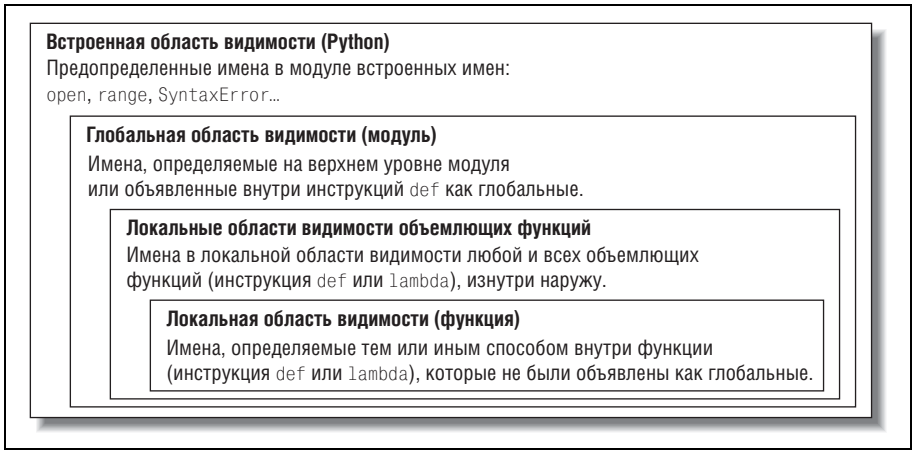

Рис. 16.1. *Правило LEGB поиска имен в областях видимости. Когда производится обращение к переменной, интерпретатор Python начинает искать ее в следующее порядке: в локальной области видимости, во всех локальных областях видимости объемлющих функций, в глобальной области видимости и, наконец, во встроенной области видимости. Поиск завершается, как только будет найдено первое подходящее имя. Место, где в программном коде производится присваивание значения переменной, обычно определяет ее область видимости*

[1] В первом издании этой книги правило поиска в областях видимости было названо «правилом LGB». Уровень объемлющей инструкции `def` был добавлен в язык Python позднее, чтобы ликвидировать необходимость явной передачи объемлющей области видимости. Но эта тема едва ли представляет интерес для начинающих, поэтому мы рассмотрим ее позднее в этой главе.

Кроме того, имейте в виду, что эти правила применяются только к простым именам *переменных* (таким как `spam`). В пятой и шестой частях книги мы увидим, что полные имена *атрибутов* (такие как `object.spam`) принадлежат определенным объектам и к ним применяются иные правила поиска, отличные от правил поиска в областях видимости, которые мы только что рассмотрели. При обращении к атрибутам (имя, следующее за точкой) поиск производится в одном или более объектах, а не в областях видимости, что связано с механизмом, который называется «наследованием» (рассматривается в шестой части книги).

Пример области видимости

Рассмотрим более крупный пример, демонстрирующий суть областей видимости. Предположим, что следующий фрагмент составляет содержимое файла модуля:

```
# Глобальная область видимости
X = 99              # X и func определены в модуле: глобальная область

def func(Y):        # Y и Z определены в функции: локальная область
    # Локальная область видимости
    Z = X + Y       # X - глобальная переменная
    return Z

func(1)             # func в модуле: result=100
```

В этом примере функция и модуль используют в своей работе несколько имен. Применяя правила области видимости языка Python, можно классифицировать эти имена следующим образом:

Глобальные имена: X и func

`X` — это глобальное имя, так как она объявлена на верхнем уровне модуля. К ней можно обращаться внутри функции, не объявляя ее глобальной. `func` — это глобальное имя по тем же причинам. Инструкция `def` связывает объект функции с именем `func` на верхнем уровне модуля.

Локальные имена: Y и Z

Имена `Y` и `Z` являются локальными (и существуют только во время выполнения функции), потому что присваивание значений обеим именам осуществляется внутри определения функции: присваивание переменной `Z` производится с помощью инструкции `=`, а `Y` — потому что аргументы всегда передаются через операцию присваивания.

Суть такого разделения имен заключается в том, что локальные переменные играют роль временных имен, которые необходимы только на время исполнения функции. Например, в предыдущем примере аргумент `Y` и результат сложения `Z` существуют только внутри функции — эти имена не пересекаются с вмещающим пространством имен модуля (или с пространствами имен любых других функций).

Разделение имен на глобальные и локальные также облегчает понимание функций, так как большинство имен, используемых в функции, появляются непосредственно в самой функции, а не в каком-то другом, произвольном месте внутри модуля. Кроме того, можно быть уверенным, что локальные имена не будут изменены любой другой удаленной функцией в программе, а это в свою очередь упрощает отладку программ.

Встроенная область видимости

Встроенная область видимости, уже упоминавшаяся выше, немного проще, чем можно было бы подумать. В действительности, встроенная область видимости – это всего лишь встроенный модуль с именем `__builtin__`, но для того, чтобы использовать имя `builtin`, необходимо импортировать модуль `__builtin__`, потому что это имя само по себе не является встроенным.

Я вполне серьезен! Встроенная область видимости реализована как модуль стандартной библиотеки с именем `__builtin__`, но само имя не находится во встроенной области видимости, поэтому, чтобы исследовать его, необходимо импортировать модуль. После этого можно будет воспользоваться функцией `dir`, чтобы получить список предопределенных имен:

```
>>> import __builtin__
>>> dir(__builtin__)
['ArithmeticError', 'AssertionError', 'AttributeError',
'DeprecationWarning', 'EOFError', 'Ellipsis',
    ...множество других имен опущено...
'str', 'super', 'tuple', 'type', 'unichr', 'unicode',
'vars', 'xrange', 'zip']
```

Имена в этом списке составляют встроенную область видимости языка Python. Примерно первая половина списка – это встроенные исключения, а вторая – встроенные функции. Согласно правилу LEGB интерпретатор выполняет поиск имен в этом модуле в последнюю очередь. Все имена из этого списка вы получаете в свое распоряжение по умолчанию, т. е., чтобы их использовать, не требуется импортировать какие-либо модули. Благодаря этому существует два способа вызвать встроенную функцию – используя правило LEGB или импортируя модуль `__builtin__` вручную:

```
>>> zip                          # Обычный способ
<встроенная функция zip>
>>> import __builtin__           # Более сложный способ
>>> __builtin__.zip
<встроенная функция zip>
```

Второй способ иногда удобно использовать при выполнении сложных действий. Внимательный читатель может заметить, что согласно правилу LEGB поиск имени прекращается, когда будет найдено первое

подходящее имя, откуда следует, что имена в локальной области видимости могут переопределять переменные с теми же самыми именами, как в глобальной, так и во встроенной области видимости, а глобальные имена могут переопределять имена во встроенной области видимости. Например, внутри функции можно создать переменную с именем `open`:

```
def hider():
    open = 'spam'      # Локальная переменная, переопределяет встроенное имя
    ...
    open('data.txt')   # В этой области видимости файл не будет открыт!
```

Однако в результате этого встроенная функция с именем `open`, которая располагается во встроенной области видимости, окажется скрытой. Обычно это считается ошибкой, и самое неприятное, что интерпретатор Python не выведет сообщения с предупреждением (иногда в программировании возникают ситуации, когда действительно бывает необходимо подменить встроенные имена, переопределив их в своем коде).[1]

Таким же способом функции могут переопределять имена глобальных переменных, определяя локальные переменные с теми же именами:

```
X = 88                 # Глобальная переменная X

def func():
    X = 99             # Локальная переменная X: переопределяет глобальную

func()
print X                # Выведет 88: значение не изменилось
```

В этом примере операция присваивания создает локальную переменную `X`, которая совершенно отлична от глобальной переменной `X`, определенной в модуле, за пределами функции. Вследствие этого внутри функции нет никакой возможности изменить переменную, расположенную за пределами функции, если не добавить объявление `global` в инструкцию `def` (как описано в следующем разделе).

---

[1] Вот вам еще один пример того, что допустимо в языке Python, но чего не следует делать: имена `True` и `False` — это всего лишь переменные во встроенной области видимости и вполне возможно их переопределить с помощью такой инструкции: `True = False`. При этом вы не нарушите общую логическую целостность! Эта инструкция всего лишь переопределит значение слова `True` в единственной области видимости. Что еще интереснее, можно выполнить даже такую инструкцию: `__builtin__.True = False`, и тогда истина станет ложью для всей программы! Такая возможность в будущем, скорее всего, будет ликвидирована (кстати, она отправляет среду IDLE в странное состояние, когда пользовательский процесс сбрасывается). Однако такой прием удобен для создателей инструментальных средств, которые вынуждены переопределять встроенные имена, такие как `open`, для нужд специализированных функций. Кроме того, следует отметить, что инструменты сторонних производителей, такие как PyChecker, выводят предупреждения о типичных ошибках программирования, включая случайное переопределение встроенных имен (эта возможность, встроенная в PyChecker, известна как «shadowing» (сокрытие)).

Инструкция global

Инструкция `global` – это единственное, что в языке Python отдаленно напоминает инструкцию объявления. Однако это не объявление типа или размера – это объявление пространства имен. Она сообщает интерпретатору, что функция будет изменять одно или более глобальных имен, т. е. имен, которые находятся в области видимости (в пространстве имен) вмещающего модуля. Инструкция `global` уже упоминалась выше, а ниже приводится общая информация о ней:

- Глобальные имена – это имена, которые определены на верхнем уровне вмещающего модуля.
- Глобальные имена должны объявляться, только если им будут присваиваться значения внутри функций.
- Обращаться к глобальным именам внутри функций можно и без объявления их глобальными.

Инструкция `global` состоит из слова `global` и следующих за ним одного или более имен, разделенных запятыми, которые будут отображены на область видимости вмещающего модуля при обращении к ним или при выполнении операции присваивания внутри тела функции. Например:

```
X = 88          # Глобальная переменная X

def func():
    global X
    X = 99      # Глобальная переменная X: за пределами инструкции def

func()
print X         # Выведет 99
```

В этом примере было добавлено объявление `global`, поэтому имя `X` внутри инструкции `def` теперь ссылается на переменную `X` за ее пределами. На этот раз оба имени представляют одну и ту же переменную. Ниже приводится более сложный пример использования инструкции `global`:

```
y, z = 1, 2             # Глобальные переменные в модуле

def all_global():
    global x            # Объявляется глобальной для присваивания
    x = y + z     # Объявлять y, z не требуется: применяется правило LEGB
```

Здесь все три переменные `x`, `y` и `z`, используемые внутри функции `all_global`, являются глобальными. Переменные `y` и `z` глобальными считаются потому, что внутри функции им не присваиваются значения. Переменная `x` считается глобальной потому, что она перечислена в инструкции `global`, которая явно отображает ее в область видимости модуля. Без инструкции `global` переменная `x` считалась бы локальной, так как ей присваивается значение внутри функции.

Обратите внимание: переменные `y` и `z` не были объявлены как глобальные, однако, следуя правилу LEGB, интерпретатор автоматически отыщет их в области видимости модуля. Кроме того, следует отметить,

что переменная x может не существовать в модуле на момент вызова функции – в этом случае операция присваивания в функции создаст переменную x в области видимости модуля.

Минимизируйте количество глобальных переменных

По умолчанию имена, значения которым присваиваются внутри функций, являются локальными, поэтому, если необходимо изменять имена за пределами функций, необходимо использовать инструкцию global. Это сделано в соответствии с общей идеологией языка Python – чтобы сделать что-то «неправильное», необходимо писать дополнительный программный код. Иногда бывает удобно использовать глобальные переменные, однако по умолчанию, если переменной присваивается значение внутри инструкции def, она становится локальной, потому что это, как правило, наилучшее решение. Изменение глобальных переменных может привести к появлению проблем, хорошо известных в разработке программного обеспечения: когда значения переменных зависят от порядка, в каком вызываются функции, это может осложнить отладку программы.

Рассмотрим следующий пример модуля:

```
X = 99

def func1():
    global X
    X = 88

def func2():
    global X
    X = 77
```

Теперь представим, что перед нами стоит задача модифицировать этот модуль или использовать его в другой программе. Каким будет значение переменной X? На самом деле этот вопрос не имеет смысла, если не указывать момент времени – значение переменной X зависит от выбранного момента времени, так как оно зависит от того, какая функция вызывалась последней (этого нельзя сказать только по одному файлу модуля).

В результате, чтобы понять этот программный код, необходимо знать путь потока выполнения всей программы. И если возникнет необходимость изменить этот модуль или использовать его в другой программе, необходимо будет удерживать в своей памяти всю программу. В этой ситуации невозможно использовать одну функцию, не принимая во внимание другую. От них зависит значение глобальной переменной. Это типичная проблема глобальных переменных – они вообще делают программный код более сложным для понимания и использования, в отличие от кода, состоящего только из независимых функций, логика выполнения которых построена на использовании локальных имен.

С другой стороны, за исключением случаев использования классов и принципов объектно-ориентированного программирования, глобальные переменные являются едва ли не самым удобным способом хранения информации о состоянии (информации, которую необходимо хранить между вызовами функции) – локальные переменные исчезают, когда функция возвращает управление, а глобальные – нет. Это можно реализовать с помощью других приемов, таких как использование изменяемых аргументов по умолчанию и области видимости объемлющих функций, но они слишком сложны по сравнению с глобальными переменными.

Некоторые программы определяют отдельный глобальный модуль для хранения всех глобальных имен – если это предусмотрено заранее, это не так вредно. Кроме того, программы на языке Python, использующие многопоточную модель выполнения для параллельной обработки данных, тесно связаны с глобальными переменными – они играют роль памяти, совместно используемой функциями, исполняющимися в параллельных потоках, и выступают в качестве средств связи (описание многопоточной модели выполнения выходит далеко за рамки данной книги, поэтому за дополнительной информацией обращайтесь к книгам, упомянутым в предисловии).

А пока, особенно если вы не имеете достаточного опыта программирования, по мере возможности избегайте искушения использовать глобальные переменные (старайтесь организовать обмен данными через параметры и возвращаемые значения). Шесть месяцев спустя вы и ваши коллеги будете рады, что поступали таким образом.

Минимизируйте количество изменений в соседних файлах

В этом разделе описывается еще одна проблема, связанная с областями видимости: несмотря на то, что *существует возможность* непосредственно изменять переменные в другом файле, этого следует избегать. Рассмотрим следующие два модуля:

```
# first.py
X = 99

# second.py
import first
first.X = 88
```

Первый модуль определяет переменную `X`, а второй – изменяет ее значение в инструкции присваивания. Обратите внимание, что для этого во втором модуле необходимо импортировать первый модуль. Как вы уже знаете, каждый модуль представляет собой отдельное пространство имен (где размещаются переменные), поэтому, чтобы увидеть содержимое одного модуля во втором, его необходимо импортировать. В терминах этой главы глобальная область видимости модуля после импор-

тирования *превращается* в пространство имен атрибутов объекта модуля – импортирующий модуль автоматически получает доступ ко всем глобальным переменным импортируемого модуля, поэтому при импортировании глобальная область видимости модуля, по сути, трансформируется в пространство имен атрибутов.

После импортирования первого модуля второй модуль присваивает его переменной новое значение. Проблема состоит в том, что эта операция выполняется слишком неявно: для любого, кто занимается сопровождением или использует первый модуль, будет сложно догадаться, что какой-то другой модуль, далеко отстоящий в цепочке импорта, может изменить значение переменной `X`. В конце концов, второй модуль может находиться вообще в другом каталоге, из-за чего его сложно будет найти. Здесь также устанавливается слишком тесная зависимость между этими двумя модулями, потому что оба они зависят от значения переменной `X`, будет трудно понять или повторно использовать один модуль без другого.

И этом случае лучшая рекомендация – не использовать такую возможность; лучше организовать взаимодействие между модулями через вызовы функций, передавая им аргументы и получая возвращаемые значения. В данном конкретном случае было бы лучше добавить функцию доступа, которая будет выполнять изменения:

```
# first.py
X = 99
def setX(new):
    global X
    X = new

# second.py
import first
first.setX(88)
```

Для этого потребуется добавить дополнительный программный код, но он имеет огромное значение в смысле обеспечения удобочитаемости и удобства в сопровождении – когда тот, кто впервые будет знакомиться с модулем, увидит функцию, он будет знать, что это часть интерфейса модуля и поймет, что переменная `X` может изменяться. Мы не можем полностью избавиться от изменений в соседних файлах, однако здравый смысл диктует необходимость минимизировать их число, если это не является широко распространенным явлением в программе.

Другие способы доступа к глобальным переменным

Интересно, что благодаря трансформации глобальных переменных в атрибуты объекта загруженного модуля существует возможность имитировать инструкцию `global`, импортируя вмещающий модуль и выполняя присваивание его атрибутам, как показано в следующем примере модуля. Программный код в этом файле в одном случае импортирует вмещающий модуль по имени, а в другом использует таблицу загру-

женных модулей `sys.modules` (подробнее об этой таблице рассказывается в главе 21):

```
# thismod.py

var = 99                        # Глобальная переменная == атрибут модуля

def local():
    var = 0                         # Изменяется локальная переменная

def glob1():
    global var                      # Глобальное объявление (обычное)
    var += 1                        # Изменяется глобальная переменная

def glob2():
    var = 0                         # Изменяется локальная переменная
    import thismod                  # Импорт самого себя
    thismod.var += 1                # Изменяется глобальная переменная

def glob3():
    var = 0                         # Изменяется локальная переменная
    import sys                      # Импорт системной таблицы
    glob = sys.modules['thismod']   # Получить объект модуля
                                    # (или использовать __name__)
    glob.var += 1                   # Изменяется глобальная переменная

def test():
    print var
    local(); glob1(); glob2(); glob3()
    print var
```

После запуска будут добавлены 3 глобальные переменные (только первая функция ничего не добавляет):

```
>>> import thismod
>>> thismod.test()
99
102
>>> thismod.var
102
```

Этот пример иллюстрирует эквивалентность глобальных имен и атрибутов модуля, однако, чтобы явно выразить свои намерения, нам потребовалось написать немного больше, чем при использовании инструкции `global`.

Области видимости и вложенные функции

До сих пор мы не еще рассматривали одну часть правила области видимости в языке Python (просто потому, что с нею редко сталкиваются на практике). Однако пришло время более пристально посмотреть на *E* в правиле LEGB. Уровень *E* появился относительно недавно (он был добавлен в Python 2.2) – это локальные области видимости объемлющих инструкций `def`. Иногда объемлющие области видимости называ-

ют *статически вложенными областями видимости*. В действительности вложение является лексическим – вложенные области видимости соответствуют физически вложенным блокам программного кода в исходных текстах программы.

В Python 3.0 предполагается появление инструкции `nonlocal`, которая позволит получать доступ на запись к переменным в областях видимости объемлющих функций, так же, как нынешняя инструкция `global` позволяет получить доступ на запись к области видимости объемлющего модуля. Синтаксически эта инструкция будет выглядеть так же, как инструкция `global`, только в ней будет использоваться слово `nonlocal`. Однако это в будущем, поэтому дополнительную информацию ищите в примечаниях к выпуску 3.0.

Вложенные области видимости

С появлением областей видимости вложенных функций правила поиска переменных стали немного более сложными. Внутри функции:

- Операция присваивания (`X = value`) создает или изменяет имя `X` в текущей локальной области видимости по умолчанию. Если имя `X` объявлено глобальным внутри функции, операция присваивания создает или изменяет имя `X` в области видимости объемлющего модуля.
- При обращении к переменной (`X`) поиск имени `X` сначала производится в локальной области видимости (функции); затем в локальных областях видимости всех лексически объемлющих функций, изнутри наружу; затем в текущей глобальной области видимости (в модуле); и, наконец, во встроенной области видимости (модуль `__builtin__`). Поиск имен, объявленных в инструкции `global`, начинается сразу с глобальной области видимости.

Обратите внимание, что инструкция `global` отображает имена в область видимости объемлющего модуля. Когда имеются вложенные функции, можно получить значения переменных в объемлющих функциях, но их нельзя изменить. Чтобы пояснить эти положения, рассмотрим их на примере программного кода.

Примеры вложенных областей видимости

Ниже приводится пример вложенной области видимости:

```
def f1():
    x = 88
    def f2():
        print x
    f2()

f1()           # Выведет 88
```

Прежде всего – это вполне допустимый программный код на языке Python: инструкция `def` – это обычная исполняемая инструкция, которая

может появляться в любом месте программы, где могут появляться любые другие инструкции, включая вложение в другую инструкцию `def`. В этом примере вложенная инструкция `def` исполняется в момент вызова функции `f1` – она создает функцию и связывает ее с именем `f2`, которое является локальным и размещается в локальной области видимости функции `f1`. В некотором смысле `f2` – это временная функция, которая существует только во время работы (и видима только для программного кода) объемлющей функции `f1`.

Однако, обратите внимание, что происходит внутри функции `f2`: когда производится вывод переменной `x`, она ссылается на переменную `x` в локальной области видимости объемлющей функции `f1`. Функции имеют возможность обращаться к именам, которые физически располагаются в любых объемлющих инструкциях `def`, и имя `x` в функции `f2` автоматически отображается на имя `x` в функции `f1` в соответствии с правилом поиска LEGB.

Это правило поиска в объемлющих областях видимости выполняется, даже если объемлющая функция фактически уже вернула управление. Например, следующий фрагмент определяет функцию, которая создает и возвращает другую функцию:

```
def f1():
    x = 88
    def f2():
        print x
    return f2

action = f1()  # Создает и возвращает функцию
action()       # Вызов этой функции: выведет 88
```

В этом фрагменте при вызове `action` фактически запускается функция, созданная во время исполнения функции `f1`. Функция `f2` помнит переменную `x` в области видимости объемлющей функции `f1`, которая уже неактивна.

Фабричные функции

В зависимости от того, кому задается вопрос о том, как называется такое поведение, можно услышать такие термины, как *замыкание* или *фабричная* функция, – объект функции, который сохраняет значения в объемлющих областях видимости, даже тогда, когда эти области могут прекратить свое существование. Классы (описываются в шестой части книги) обычно лучше подходят для сохранения состояния, потому что они позволяют делать это явно, посредством присваивания значений атрибутам, тем не менее, подобные функции обеспечивают другую альтернативу.

Например, фабричные функции иногда используются в программах, когда необходимо создавать обработчики событий прямо в процессе исполнения, в соответствии со сложившимися условиями (например,

когда желательно запретить пользователю вводить данные). Рассмотрим в качестве примера следующую функцию:

```
>>> def maker(N):
...     def action(X):
...         return X ** N
...     return action
...
```

Здесь определяется внешняя функция, которая просто создает и возвращает вложенную функцию, не вызывая ее. Если вызвать внешнюю функцию:

```
>>> f = maker(2)                # Запишет 2 в N
>>> f
<function action at 0x014720B0>
```

она вернет ссылку на созданную ею внутреннюю функцию, созданную при выполнении вложенной инструкции `def`. Если теперь вызвать то, что было получено от внешней функции:

```
>>> f(3)       # Запишет 3 в X, в N по-прежнему хранится число 2
9
>>> f(4)       # 4 ** 2
16
```

будет вызвана вложенная функция, с именем `action` внутри функции `maker`. Самое необычное здесь то, что вложенная функция продолжает хранить число 2, значение переменной `N` в функции `maker` даже при том, что к моменту вызова функции `action` функция `maker` уже завершила свою работу и вернула управление. В действительности имя `N` из объемлющей локальной области видимости сохраняется как информация о состоянии, присоединенная к функции `action`, и мы получаем обратно значение аргумента, возведенное в квадрат.

Теперь, если снова вызвать внешнюю функцию, мы получим новую вложенную функцию уже с другой информацией о состоянии, присоединенной к ней, – в результате вместо квадрата будет вычисляться куб аргумента, но ранее сохраненная функция по-прежнему будет возвращать квадрат аргумента:

```
>>> g = maker(3)
>>> g(3)               # 3 ** 3
27
>>> f(3)               # 3 ** 2
9
```

Это довольно сложный прием, который вам вряд ли часто придется часто встречать на практике, впрочем, он распространен среди программистов, обладающих опытом работы с функциональными языками программирования (и иногда его можно встретить в выражениях `lambda`, как будет описано далее). Вообще классы, которые будут обсуждаться позднее, лучше подходят на роль «памяти», как в данном случае, пото-

му что они обеспечивают явное сохранение информации. Помимо классов основными средствами хранения информации о состоянии функций в языке Python являются глобальные переменные, объемлющие области видимости, как в данном случае, и аргументы по умолчанию.

Сохранение состояния объемлющей области видимости с помощью аргументов по умолчанию

В предыдущих версиях Python такой программный код, как в предыдущем разделе, терпел неудачу из-за отсутствия вложенных областей видимости в инструкциях `def` – при обращении к переменной внутри функции `f2` поиск производился сначала в локальной области видимости (`f2`), затем в глобальной (программный код за пределами `f1`) и затем во встроенной области видимости. Области видимости объемлющих функций не просматривались, что могло приводить к ошибке. Чтобы разрешить ситуацию, программисты обычно использовали значения аргументов по умолчанию для передачи (сохранения) объектов, расположенных в объемлющей области видимости:

```
def f1():
    x = 88
    def f2(x=x):
        print x
    f2()

f1()                    # Выведет 88
```

Этот фрагмент будет работать во всех версиях Python, и такой подход по-прежнему можно встретить в существующих программах на языке Python. Аргументы со значениями по умолчанию мы рассмотрим ниже, в этой же главе. А пока в двух словах замечу, что конструкция `arg = val` в заголовке инструкции `def` означает, что аргумент `arg` по умолчанию будет иметь значение `val`, если функции не передается какого-либо другого значения.

В измененной версии `f2` запись `x=x` означает, что аргумент `x` по умолчанию будет иметь значение переменной `x` объемлющей области видимости. Поскольку значение для второго имени `x` вычисляется еще до того, как интерпретатор Python войдет во вложенную инструкцию `def`, оно все еще ссылается на имя `x` в функции `f1`. В результате в значении по умолчанию запоминается значение переменной `x` в функции `f1` (т. е. объект `88`).

Все это довольно сложно и полностью зависит от того, когда вычисляется значение по умолчанию. Фактически поиск во вложенных областях видимости был добавлен в Python, чтобы избавиться от такого способа использования значений по умолчанию – сейчас Python автоматически сохраняет любые значения в объемлющей области видимости для последующего использования во вложенных инструкциях `def`.

Безусловно, наилучшей рекомендацией будет просто избегать вложения инструкций `def` в другие инструкции `def`, так как это существенно

упростит программы. Ниже приводится фрагмент, который является эквивалентом предшествующего примера, в котором просто отсутствует понятие вложенности. Обратите внимание, что вполне допустимо вызывать функцию, определение которой в тексте программы находится ниже функции, откуда производится вызов, как в данном случае, при условии, что вторая инструкция `def` будет исполнена до того, как первая функция попытается вызвать ее, – программный код внутри инструкции `def` не выполняется, пока не будет произведен фактический вызов функции:

```
>>> def f1():
...     x = 88
...     f2(x)
...
>>> def f2(x):
...    print x
...
>>> f1()
88
```

При использовании такого способа можно забыть о концепции вложенных областей видимости в языке Python, если вам не потребуется создавать фабричные функции, обсуждавшиеся выше, – по крайней мере при использовании инструкций `def`. Выражения `lambda`, которые практически всегда вкладываются в инструкции `def`, часто используют вложенные области видимости, как описывается в следующем разделе.

Вложенные области видимости и lambda-выражения

Несмотря на то, что вложенные области видимости на практике редко используются непосредственно для инструкций `def`, вам наверняка придется столкнуться с областями видимости вложенных функций, когда вы начнете использовать выражения `lambda`. Мы не будем подробно рассматривать эти выражения до главы 17, но в двух словах замечу, что это выражение генерирует новую функцию, которая будет вызываться позднее, и оно очень похоже на инструкцию `def`. Поскольку `lambda` – это выражение, оно может использоваться там, где не допускается использование инструкции `def`, например, в литералах списков и словарей.

Подобно инструкции `def`, выражение `lambda` сопровождается появлением новой локальной области видимости. Благодаря наличию возможности поиска имен в объемлющей области видимости выражения `lambda` способны обращаться ко всем переменным, которые присутствуют в функциях, где находятся эти выражения. Таким образом, в настоящее время следующий программный код будет работать исключительно благодаря тому, что в настоящее время действуют правила поиска во вложенных областях видимости:

```
def func():
    x = 4
```

```
    action = (lambda n: x ** n)  # Запоминается x из объемлющей
                                 # инструкции def
    return action

x = func()
print x(2)                       # Выведет 16, 4 ** 2
```

До того как появилось понятие областей видимости вложенных функций, программисты использовали значения по умолчанию для передачи значений из объемлющей области видимости в выражения `lambda` точно так же, как и в случае с инструкциями `def`. Например, следующий фрагмент будет работать во всех версиях Python:

```
def func():
    x = 4
    action = (lambda n, x=x: x ** n)    # Передача x вручную
```

Поскольку `lambda` – это выражения, они естественно (и даже обычно) вкладываются внутрь инструкций `def`. Следовательно, именно они извлекли наибольшую выгоду от добавления областей видимости объемлющих функций в правила поиска имен – в большинстве случаев отпадает необходимость передавать в выражения `lambda` аргументы со значениями по умолчанию.

Области видимости и значения по умолчанию применительно к переменным цикла

Существует одно известное исключение из правила, которое я только что дал: если `lambda`-выражение или инструкция `def` вложены в цикл внутри другой функции и вложенная функция ссылается на переменную из объемлющей области видимости, которая изменяется в цикле, все функции, созданные в этом цикле, будут иметь одно и то же значение – значение, которое имела переменная на последней итерации.

Например, ниже предпринята попытка создать список функций, каждая из которых запоминает текущее значение переменной `i` из объемлющей области видимости:

```
>>> def makeActions():
...     acts = []
...     for i in range(5):                 # Сохранить каждое значение i
...         acts.append(lambda x: i ** x)  # Все запомнят последнее значение i!
...     return acts
...
>>> acts = makeActions()
>>> acts[0]
<function <lambda> at 0x012B16B0>
```

Такой подход не дает желаемого результата, потому что поиск переменной в объемлющей области видимости производится позднее, при вызове вложенных функций, в результате все они получат одно и то же значение (значение, которое имела переменная цикла на последней итерации). То есть каждая функция в списке будет возвращать 4 во

второй степени, потому что во всех них переменная i имеет одно и то же значение:

```
>>> acts[0](2)     # Все возвращают 4 ** 2, последнее значение i
16
>>> acts[2](2)     # Здесь должно быть 2 ** 2
16
>>> acts[4](2)     # Здесь должно быть 4 ** 2
16
```

Это один из случаев, когда необходимо явно сохранять значение из объемлющей области видимости в виде аргумента со значением по умолчанию вместо использования ссылки на переменную из объемлющей области видимости. То есть, чтобы этот фрагмент заработал, необходимо передать текущее значение переменной из объемлющей области видимости в виде значения по умолчанию. Значения по умолчанию вычисляются в момент создания вложенной функции (а не когда она вызывается), поэтому каждая из них сохранит свое собственное значение i:

```
>>> def makeActions():
...     acts = []
...     for i in range(5):                # Использовать значения по умолчанию
...     acts.append(lambda x, i=i: i ** x)   # Сохранить текущее значение i
...     return acts
...
>>> acts = makeActions()
>>> acts[0](2)                            # 0 ** 2
0
>>> acts[2](2)                            # 2 ** 2
4
>>> acts[4](2)                            # 4 ** 2
16
```

Это достаточно замысловатый случай, но с ним можно столкнуться на практике, особенно в программном коде, который генерирует функции-обработчики событий для элементов управления в графическом интерфейсе (например, обработчики нажатия кнопок). Подробнее о значениях по умолчанию и lambda-выражениях мы поговорим в следующей главе, поэтому позднее вам может потребоваться вернуться к этому разделу.[1]

[1] В разделе «Ошибки при работе с функциями» к этой части книги, в конце следующей главы, мы также увидим, что существуют определенные проблемы с использованием изменяемых объектов, таких как списки и словари, при использовании их в качестве значений по умолчанию для аргументов (например, def f(a=[])). Так как значения по умолчанию реализованы в виде единственных объектов, изменяемые объекты по умолчанию сохраняют свое состояние от вызова к вызову, а не инициализируются заново при каждом вызове. В зависимости от того, кому задается вопрос, эта особенность может рассматриваться или как особенность, поддерживающая возможность сохранения состояния, или как недостаток языка. Подробнее об этом будет говориться в следующей главе.

Произвольное вложение областей видимости

Прежде чем закончить это исследование, я должен заметить, что области видимости могут вкладываться произвольно, но поиск будет производиться только в объемлющих функциях (не в классах, которые описываются в шестой части книги):

```
>>> def f1():
...     x = 99
...     def f2():
...         def f3():
...             print x        # Будет найдена в области видимости f1!
...         f3()
...     f2()
...
>>> f1()
99
```

Интерпретатор будет искать переменную в локальных областях видимости *всех* объемлющих инструкций `def`, начиная от внутренних к внешним, выше локальной области видимости и ниже глобальной области видимости модуля. Однако такой программный код едва ли может получиться на практике. В языке Python считается, что плоское лучше вложенного – ваша жизнь и жизнь ваших коллег будет проще, если вы сведете к минимуму количество вложенных определений функций.

Передача аргументов

Раньше уже говорилось, что передача аргументов производится посредством операции присваивания. Здесь имеется несколько моментов, не всегда очевидных для начинающих, о которых будет говориться в этом разделе. Ниже приводится несколько важных замечаний, касающихся передачи аргументов в функции:

- **Аргументы передаются через автоматическое присваивание объектов локальным именам.** Аргументы функции – ссылки на объекты, которые (возможно) используются совместно с вызывающей программой – это всего лишь результат еще одной из разновидностей операции присваивания. Ссылки в языке Python реализованы в виде указателей, поэтому все аргументы фактически передаются по указателям. Объекты, которые передаются в виде аргументов, никогда не копируются.
- **Операция присваивания именам аргументов внутри функции не оказывает влияния на вызывающую программу.** При вызове функции имена аргументов, указанные в ее заголовке, становятся новыми локальными именами в области видимости функции. Это не является совмещением имен между именами аргументов и именами в вызывающей программе.

- **Изменение внутри функции аргумента, который является изменяемым объектом, может оказывать влияние на вызывающую программу.** С другой стороны, так как аргументы – это всего лишь результат операции присваивания полученных объектов, функции могут воздействовать на полученные изменяемые объекты и тем самым оказывать влияние на вызывающую программу. Изменяемые объекты могут рассматриваться функциями, как средство ввода, так и вывода информации.

За дополнительной информацией о ссылках обращайтесь к главе 6 – все, что там говорится, вполне применимо и к аргументам функций несмотря на то, что присваивание именам аргументов выполняется автоматически и неявно.

Схема передачи аргументов посредством присваивания, принятая в языке Python, это далеко не то же самое, что передача аргументов по ссылке в языке C++, но она очень близка к модели передачи аргументов в языке C:

- **Неизменяемые объекты передаются «по значению».** Такие объекты, как целые числа и строки, передаются в виде ссылок на объекты, а не в виде копий объектов, но, так как неизменяемые объекты невозможно изменить непосредственно, передача таких объектов очень напоминает копирование.
- **Изменяемые объекты передаются «по указателю».** Такие объекты, как списки и словари, также передаются в виде ссылок на объекты, что очень похоже на то, как в языке C передаются указатели на массивы – изменяемые объекты допускают возможность непосредственного изменения внутри функции так же, как и массивы в языке C.

Конечно, если вы никогда ранее не использовали язык C, модель передачи аргументов в языке Python буде казаться вам еще проще – согласно этой модели выполняется присваивание объектов именам аргументов, и она одинаково работает с любыми объектами, как с изменяемыми, так и с неизменяемыми.

Аргументы и разделяемые ссылки

Ниже приводится пример, который иллюстрирует работу этих положений:

```
>>> def changer(a, b):   # Функция
...     a = 2            # Изменяется только значение локального имени
...     b[0] = 'spam'    # Изменяется непосредственно разделяемый объект
...
>>> X = 1
>>> L = [1, 2]           # Вызывающая программа
>>> changer(X, L)        # Передаются изменяемый и неизменяемый объекты
>>> X, L                 # X - не изменилась, L - изменилась
(1, ['spam', 2])
```

В этом фрагменте функция changer присваивает значения аргументу a и компоненту объекта, на который ссылается аргумент b. Две операции присваивания в функции имеют незначительные синтаксические различия, но дают совершенно разные результаты:

- Так как a – это локальное имя в области видимости функции, первая операция присваивания не имеет эффекта для вызывающей программы – она всего лишь изменяет локальную переменную a и не изменяет связанное с ней имя X в вызывающей программе.
- b – также локальное имя, но в нем передается изменяемый объект (список, с именем L в вызывающей программе). Поскольку вторая операция присваивания воздействует непосредственно на изменяемый объект, результат присваивания элементу b[0] в функции оказывает влияние на значение имени L после выхода из функции. В действительности эта операция не изменяет объект b, она изменяет часть объекта, на который ссылается аргумент b, и это изменение оказывает влияние на вызывающую программу.

Рис. 16.2 иллюстрирует связи имя/объект, которые имеют место непосредственно сразу после вызова функции, но перед тем, как будет запущено ее тело.

Если этот пример все еще кажется вам непонятным, попробуйте представить себе автоматическое присваивание переданным аргументам,

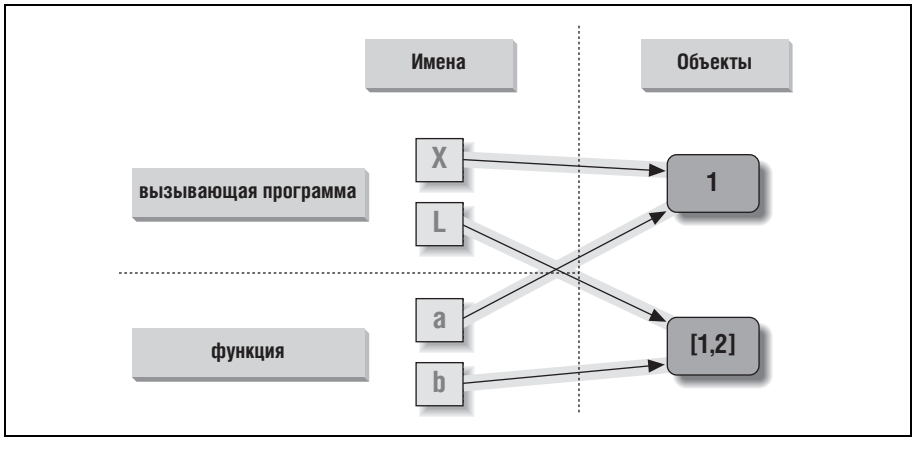

Рис. 16.2. *Ссылки: аргументы. Так как аргументы передаются посредством операции присваивания, имена аргументов могут ссылаться на объекты, на которые ссылаются переменные, участвующие в вызове функции. Следовательно, непосредственные воздействия на изменяемые аргументы внутри функции могут оказывать влияние на вызывающую программу. В данном случае a и b внутри функции изначально ссылаются на те же объекты, на которые ссылаются переменные X и L при первом вызове функции. Изменение списка, выполненное через переменную b, можно наблюдать в переменной L после того, как функция вернет управление*

как последовательность простых инструкций присваивания. Для первого аргумента операции присваивания не оказывают влияния на вызывающую программу:

```
>>> X = 1
>>> a = X      # Разделяют один и тот же объект
>>> a = 2      # Изменяется только 'a', значение 'X' остается равным 1
>>> print X
1
```

Но для второго аргумента операция присваивания сказывается на значении переменной, участвующей в вызове, так как она производит непосредственное изменение объекта:

```
>>> L = [1, 2]
>>> b = L              # Разделяют один и тот же объект
>>> b[0] = 'spam'      # Непосредственное изменение: 'L' также изменяется
>>> print L
['spam', 2]
```

Вспомните обсуждение вопросов совместного использования изменяемых объектов в главах 6 и 9, и вы узнаете это явление: непосредственное воздействие на изменяемый объект может сказываться на других ссылках на этот объект. В данном случае один из параметров функции играет роль средства вывода информации.

Как избежать воздействий на изменяемые аргументы

В языке Python по умолчанию аргументы передаются в функции по ссылкам (т. е. по указателям), потому что в большинстве случаев именно это и требуется – это означает, что мы можем передавать крупные объекты в любые точки программы без создания множества копий и легко изменять эти объекты в процессе работы. Однако, если нам требуется избежать влияния непосредственных изменений объектов, производимых в функциях, на вызывающую программу, мы можем просто явно копировать изменяемые объекты, как описывалось в главе 6. В случае с аргументами функций мы всегда можем скопировать список в точке вызова:

```
L = [1, 2]
changer(X, L[:])    # Передается копия, поэтому переменная 'L' не изменится
```

Можно также создать копию внутри функции, если нам не требуется изменять полученный объект независимо от того, как была вызвана функция:

```
def changer(a, b):
    b = b[:]        # Входной список копируется, что исключает
                    # воздействие на вызывающую программу
    a = 2
    b[0] = 'spam'   # Изменится только копия списка
```

Оба варианта копирования не мешают функции изменять объект, они лишь препятствуют воздействию этих изменений на вызывающую программу. Чтобы действительно предотвратить изменения, мы всегда можем преобразовать изменяемые объекты в неизменяемые, что позволит выявить источники проблем. Например, при попытке изменения кортежа будет возбуждено исключение:

```
L = [1, 2]
changer(X, tuple(L))  # Передается кортеж, поэтому попытка изменения
                      # возбудит исключение
```

Здесь используется встроенная функция `tuple`, которая создает новый кортеж из всех элементов последовательности (в действительности – любого итерируемого объекта). Это особенно важно, потому что вынуждает писать функцию так, чтобы она никогда не пыталась изменять передаваемые ей аргументы. Однако такое решение накладывает на функцию больше ограничений, чем следует, поэтому его вообще следует избегать. Нельзя знать заранее, когда изменение аргументов окажется необходимым. Кроме того, при таком подходе функция теряет возможность применять к аргументу методы списков, включая методы, которые не производят непосредственных изменений.

Главное, что следует запомнить, – функции могут оказывать воздействия на передаваемые им изменяемые объекты (например, на списки и словари). Это не всегда является проблемой и часто может приносить пользу. Но вам действительно необходимо знать об этой особенности – если изменения в объектах происходят неожиданно для вас, проверьте вызываемые функции и в случае необходимости передавайте им копии объектов.

Имитация выходных параметров

Мы уже обсудили инструкцию `return` и использовали ее в нескольких примерах. Эта инструкция может возвращать объект любого типа, поэтому с ее помощью можно возвращать сразу несколько значений, упаковав их в кортеж или в коллекцию любого другого типа. В языке Python фактически отсутствует такое понятие, которое в некоторых других языках называется «передача аргументов по ссылке», однако мы можем имитировать такое поведение, возвращая кортеж и выполняя присваивание результатов оригинальным именам аргументов в вызывающей программе:

```
>>> def multiple(x, y):
...     x = 2          # Изменяется только локальное имя
...     y = [3, 4]
...     return x, y    # Новые значения возвращаются в виде кортежа
...
>>> X = 1
>>> L = [1, 2]
>>> X, L = multiple(X, L)      # Результаты присваиваются именам
```

```
>>> X, L                        # в вызывающей программе
(2, [3, 4])
```

Выглядит так, как будто функция возвращает два значения, но на самом деле – это единственный кортеж, состоящий из двух элементов, а необязательные окружающие скобки просто опущены. После возврата из функции можно использовать операцию присваивания кортежа, чтобы извлечь отдельные элементы. (Если вы забыли, как это работает, вернитесь к разделу «Кортежи» в главе 4 и «Инструкции присваивания» в главе 11.) Такой прием позволяет имитировать выходные параметры, имеющиеся в других языках программирования, за счет использования явной операции присваивания. Переменные `X` и `L` изменятся после вызова функции, но только потому, что мы явно это запрограммировали.

Специальные режимы сопоставления аргументов

Как только что было показано, в языке Python аргументы всегда передаются через операцию *присваивания* – передаваемые объекты присваиваются именам, указанным в заголовке инструкции `def`. Однако на основе этой модели язык Python обеспечивает дополнительные возможности влиять на способ, которым объекты аргументов сопоставляются с именами аргументов в заголовке функции. Эти возможности можно и не использовать, но они позволяют писать функции, поддерживающие более гибкие схемы вызова.

По умолчанию сопоставление аргументов производится в соответствии с их позициями, слева направо, и функции должно передаваться столько аргументов, сколько имен указано в заголовке функции. Но кроме этого существует возможность явно указывать соответствия между аргументами и именами, определять значения по умолчанию и передавать дополнительные аргументы.

Это достаточно сложный раздел, и прежде чем окунуться в обсуждение синтаксиса, я хотел бы подчеркнуть, что эти специальные режимы не являются обязательными и имеют отношение только к сопоставлению объектов и имен – основным механизмом передачи аргументов попрежнему остается операция присваивания. Фактически некоторые из этих режимов предназначены в первую очередь для разработчиков библиотек, а не для разработчиков приложений. Но, так как вы можете столкнуться с этими режимами, даже не используя их в своих программах, я коротко опишу их:

Сопоставление по позиции: значения и имена ставятся в соответствие по порядку, слева направо

Обычный случай, который мы использовали до сих пор: значения и имена аргументов ставятся в соответствие в порядке их следования слева направо.

Сопоставление по ключам: соответствие определяется по указанным именам аргументов

Вызывающая программа имеет возможность указать соответствие между аргументами функции и их значениями в момент вызова, используя синтаксис `name=value`.

Значения по умолчанию: указываются значения аргументов, которые могут не передаваться

Функции могут определять значения аргументов по умолчанию на тот случай, если вызывающая программа передаст недостаточное количество значений. Здесь также используется синтаксис `name=value`.

Переменное число аргументов: прием произвольного числа аргументов, позиционных или в виде ключей

Функции могут использовать специальный аргумент, имени которого предшествует символ `*`, для объединения произвольного числа дополнительных аргументов в коллекцию (эта особенность часто называется *varargs*, как в языке C, где также поддерживаются списки аргументов переменной длины).

Переменное число аргументов: передача произвольного числа аргументов, позиционных или в виде ключей

Вызывающая программа также может использовать символ `*` для передачи коллекции аргументов в виде отдельных аргументов. В данном случае символ `*` имеет обратный смысл по отношению к символу `*` в заголовке функции – в заголовке он означает коллекцию произвольного числа аргументов, тогда как в вызывающей программе – передачу коллекции в виде произвольного числа отдельных аргументов.

В табл. 16.1 приводится синтаксис использования специальных режимов сопоставления.

Таблица 16.1. Виды сопоставления аргументов функций

Синтаксис	Местоположение	Интерпретация
`func(value)`	Вызывающая программа	Обычный аргумент: сопоставление производится по позиции
`func(name=value)`	Вызывающая программа	По ключу: сопоставление производится по указанному имени
`func(*name)`	Вызывающая программа	Все объекты в указанном имени передаются как отдельные позиционные аргументы
`func(**name)`	Вызывающая программа	Все пары ключ/значение в указанном имени передаются как отдельные аргументы по ключевым словам
`def func(name)`	Функция	Обычный аргумент: сопоставление производится по позиции или по имени

Синтаксис	Местоположение	Интерпретация
`def func(name=value)`	Функция	Значение аргумента по умолчанию, на случай, если аргумент не передается функции
`def func(*name)`	Функция	Определяет и объединяет все дополнительные аргументы в коллекцию (в кортеж)
`def func(**name)`	Функция	Определяет и объединяет все дополнительные аргументы по ключам (в словарь)

В вызывающей программе (первые четыре строки таблицы) при использовании простых значений соответствие именам аргументов определяется по позиции, но при использовании формы `name=value` соответствие определяется по именам аргументов – это называется *передачей аргументов по ключам*. Использование символов `*` и `**` в вызывающей программе позволяет передавать произвольное число объектов по позиции или по ключам в виде последовательностей и словарей соответственно.

В заголовке функции при использовании простых значений соответствие определяется по позиции или по имени (в зависимости от того, как вызывающая программа передает значения), но при использовании формы `name=value` определяются значения по умолчанию. При использовании формы `*name` все дополнительные аргументы объединяются в кортеж, а при использовании формы `**name` – в словарь.

Наиболее часто в программном коде на языке Python используются форма передачи аргументов по ключам и по умолчанию. Возможность передачи аргументов по ключам позволяет указывать значения аргументов вместе с их именами, чтобы придать вызову функции больше смысла. Со значениями по умолчанию мы уже встречались, когда рассматривали способы передачи значений из объемлющей области видимости, но на самом деле эта форма имеет более широкую область применения – она позволяет определять необязательные аргументы и указывать значения по умолчанию в определении функции.

Специальные режимы сопоставления позволяют обеспечить свободу выбора числа аргументов, которые должны передаваться функции в обязательном порядке. Если функция определяет значения по умолчанию, они будут использоваться, когда функции передается недостаточное число аргументов. Если функция использует форму `*` определения списка аргументов переменной длины, она сможет принимать большее число аргументов – дополнительные аргументы будут собраны в структуру данных под именем с символом `*`.

Примеры использования ключей и значений по умолчанию

Предыдущие пояснения в программном коде выглядят гораздо проще. В языке Python по умолчанию сопоставление значений и имен аргументов производится по позиции, как и в большинстве других языков. Например, если определена функция, которая требует передачи трех аргументов, она должна вызываться с тремя аргументами:

```
>>> def f(a, b, c): print a, b, c
...
```

Здесь значения передаются по позиции – имени `a` соответствует значение `1`, имени `b` соответствует значение `2` и т. д.:

```
>>> f(1, 2, 3)
1 2 3
```

Ключи

В языке Python существует возможность явно определить соответствия между значениями и именами аргументов при вызове функции. Ключи позволяют определять соответствие по *именам*, а не по позициям:

```
>>> f(c=3, b=2, a=1)
1 2 3
```

В этом вызове `c=3`, например, означает, что значение `3` передается функции в аргументе с именем `c`. Говоря более формальным языком, интерпретатор сопоставляет имя `c` в вызове функции с именем аргумента `c` в заголовке определения функции и затем передает значение `3` в этот аргумент. Результат этого вызова будет тем же, что и предыдущего. Обратите внимание, что при использовании ключей порядок следования аргументов не имеет никакого значения, потому что сопоставление производится по именам, а не по позициям. Существует даже возможность объединять передачу аргументов по позициям и по ключам в одном вызове. В этом случае сначала будут сопоставлены все позиционные аргументы, слева направо, а потом будет выполнено сопоставление ключей с именами:

```
>>> f(1, c=3, b=2)
1 2 3
```

Большинство из тех, кто впервые сталкивается с такой возможностью, задаются вопросом: где такая возможность может пригодиться? В языке Python ключи обычно играют две роли. Первая: они делают вызовы функций более описательными (представьте, что вы используете имена аргументов более осмысленные, чем простые `a`, `b` и `c`). Например, такой вызов:

```
func(name='Bob', age=40, job='dev')
```

несет больше смысла, чем вызов с тремя простыми значениями, разделенными запятыми, – ключи играют роль меток для данных, участвующих в вызове. Вторая: ключи используются вместе со значениями по умолчанию, о которых мы поговорим далее.

Значения по умолчанию

О значениях по умолчанию мы немного говорили ранее, когда обсуждали области видимости вложенных функций. Если коротко, значения по умолчанию позволяют сделать отдельные аргументы функции необязательными – если значение не передается при вызове, аргумент получит значение по умолчанию, перед тем как будет запущено тело функции. Например, ниже приводится функция, в которой один аргумент является обязательным, а два имеют значения по умолчанию:

```
>>> def f(a, b=2, c=3): print a, b, c
...
```

При вызове такой функции мы обязаны передать значение для аргумента `a`, по позиции или по ключу, а значения для аргументов `b` и `c` можно опустить. Если значения аргументов `b` и `c` будут опущены, они примут значения по умолчанию `2` и `3`, соответственно:

```
>>> f(1)
1 2 3
>>> f(a=1)
1 2 3
```

Если функции передать только два значения, аргумент `c` примет значение по умолчанию, а если три – ни одно из значений по умолчанию не будет использовано:

```
>>> f(1, 4)
1 4 3
>>> f(1, 4, 5)
1 4 5
```

Наконец, ниже приводится пример взаимодействия режимов передачи значений по ключу и по умолчанию. Поскольку ключи могут нарушать обычный порядок отображения аргументов слева направо, они, по сути, позволяют «перепрыгивать» через аргументы со значениями по умолчанию:

```
>>> f(1, c=6)
1 2 6
```

Здесь значение `1` будет сопоставлено с аргументом по позиции, аргумент `c` получит значение `6`, а аргумент `b`, в середине, – значение по умолчанию `2`.

Не путайте синтаксические конструкции `name=value` в заголовке функции и в вызове функции – в вызове она означает использование режима сопоставления значения с именем аргумента по ключу, а в заголовке

определяет значение по умолчанию для необязательного аргумента. В обоих случаях это не инструкция присваивания – это особая синтаксическая конструкция для этих двух случаев, которая модифицирует механику сопоставления значений с именами аргументов, используемую по умолчанию.

Примеры произвольного числа аргументов

Последние два расширения * и ** механизма сопоставления аргументов и их значений предназначены для поддержки возможности передачи произвольного числа аргументов функциям. Оба варианта могут появляться как в определениях функций, так и в их вызовах, и в обоих случаях они имеют сходные назначения.

Сбор аргументов в коллекцию

В первом случае, в определении функции, выполняется сборка лишних позиционных аргументов в кортеж:

```
>>> def f(*args): print args
...
```

При вызове этой функции интерпретатор Python соберет все позиционные аргументы в новый кортеж и присвоит этот кортеж переменной `args`. Это будет обычный объект кортежа, поэтому из него можно извлекать элементы по индексам, выполнять обход в цикле `for` и т. д.:

```
>>> f()
()
>>> f(1)
(1,)
>>> f(1,2,3,4)
(1, 2, 3, 4)
```

Комбинация ** дает похожий результат, но применяется при передаче аргументов по ключам – в этом случае аргументы будут собраны в новый словарь, который можно обрабатывать обычными инструментами, предназначенными для работы со словарями. В определенном смысле форма ** позволяет преобразовать аргументы, передаваемые по ключам, в словари, которые можно будет обойти с помощью метода `keys`, итераторов словарей и т. д.:

```
>>> def f(**args): print args
...
>>> f()
{ }
>>> f(a=1, b=2)
{'a': 1, 'b': 2}
```

Наконец, в заголовках функций можно комбинировать обычные аргументы, * и ** для реализации чрезвычайно гибких сигнатур вызова:

```
>>> def f(a, *pargs, **kargs): print a, pargs, kargs
```

```
...
>>> f(1, 2, 3, x=1, y=2)
1 (2, 3) {'y': 2, 'x': 1}
```

Фактически все эти формы передачи аргументов можно объединять в еще более сложные комбинации, которые на первый взгляд могут показаться неоднозначными – к этой идее мы еще вернемся ниже, в этой главе. А сейчас посмотрим, как используются формы * и ** в вызовах функций.

Извлечение аргументов из коллекции

В последних версиях Python форму * можно также использовать в вызовах функций. В этом случае данная форма передачи аргументов имеет противоположный смысл по сравнению с применением этой формы в определениях функций – она распаковывает, а не создает коллекцию аргументов. Например, можно передать в функцию четыре аргумента в виде кортежа и позволить интерпретатору распаковать их в отдельные аргументы:

```
>>> def func(a, b, c, d): print a, b, c, d
...
>>> args = (1, 2)
>>> args += (3, 4)
>>> func(*args)
1 2 3 4
```

Точно также форма ** в вызовах функций распаковывает словари пар ключ/значение в отдельные аргументы, которые передаются по ключу:

```
>>> args = {'a': 1, 'b': 2, 'c': 3}
>>> args['d'] = 4
>>> func(**args)
1 2 3 4
```

Здесь также можно очень гибко комбинировать в одном вызове обычные позиционные аргументы и аргументы по ключу:

```
>>> func(*(1, 2), **{'d': 4, 'c': 4})
1 2 4 4

>>> func(1, *(2, 3), **{'d': 4})
1 2 3 4

>>> func(1, c=3, *(2,), **{'d': 4})
1 2 3 4
```

Такого рода программный код удобно использовать, когда заранее невозможно предсказать число аргументов, которые могут быть переданы функции – вы можете собирать коллекцию аргументов во время исполнения и вызывать функцию таким способом. Не путайте синтаксические конструкции */** в заголовках функций и в их вызовах – в заголовке они означают сбор всех лишних аргументов в коллекцию, а в вызове выполняют распаковку коллекций в отдельные аргументы.

Мы еще вернемся к этим формам в следующей главе, когда будем рассматривать встроенную функцию `apply` (для замены которой, в значительной степени, предназначены эти конструкции).

Комбинирование ключей и значений по умолчанию

Ниже приводится немного больший по объему пример, демонстрирующий передачу аргументов по ключам и по умолчанию в действии. В этом примере вызывающая программа всегда должна передавать функции как минимум два аргумента (`spam` и `eggs`), два других аргумента являются необязательными. В случае их отсутствия интерпретатор присвоит именам `toast` и `ham` значения по умолчанию, указанные в заголовке:

```
def func(spam, eggs, toast=0, ham=0):   # Первые 2 являются обязательными
    print (spam, eggs, toast, ham)

func(1, 2)                              # Выведет: (1, 2, 0, 0)
func(1, ham=1, eggs=0)                  # Выведет: (1, 0, 0, 1)
func(spam=1, eggs=0)                    # Выведет: (1, 0, 0, 0)
func(toast=1, eggs=2, spam=3)           # Выведет: (3, 2, 1, 0)
func(1, 2, 3, 4)                        # Выведет: (1, 2, 3, 4)
```

Обращаю снова ваше внимание: когда в вызовах используются ключи аргументов, порядок следования аргументов не имеет значения, потому что сопоставление выполняется по именам, а не по позициям. Вызывающая программа обязана передать значения для аргументов `spam` и `eggs`, а сопоставление может выполняться как по позиции, так и по ключам. Обратите также внимание на то, что форма `name=value` имеет разный смысл в вызове функции и в инструкции `def` (ключ – в вызове и значение по умолчанию – в заголовке).

Функция поиска минимума

Чтобы придать обсуждению больше конкретики, выполним упражнение, которое демонстрирует практическое применение механизмов сопоставления аргументов. Предположим, что вам необходимо написать функцию, которая способна находить минимальное значение из произвольного множества аргументов с произвольными типами данных. То есть функция должна принимать ноль или более аргументов – столько, сколько вы пожелаете передать. Более того, функция должна работать со всеми типами объектов, имеющимися в языке Python: числами, строками, списками, списками словарей, файлами и даже `None`.

Первое требование представляет собой обычный пример того, как можно найти применение форме `*` – мы можем собирать аргументы в кортеж и выполнять их обход с помощью простого цикла `for`. Второе требование тоже не представляет никакой сложности: все типы объектов поддерживают операцию сравнения, поэтому нам не требуется учитывать типы объектов в функции (полиморфизм в действии) – мы можем просто слепо сравнивать объекты и позволить интерпретатору самостоятельно выбрать корректную операцию сравнения.

Основное задание

Ниже представлены три способа реализации этой функции, из которых, по крайней мере, один был предложен студентом:

- Первая версия функции извлекает первый аргумент (`args` – это кортеж) и обходит остальную часть коллекции, отсекая первый элемент (нет никакого смысла сравнивать объект сам с собой, особенно, если это довольная крупная структура данных).
- Вторая версия позволяет интерпретатору самому выбрать первый аргумент и остаток, благодаря чему отсутствует необходимость извлекать первый аргумент и получать срез.
- Третья версия преобразует кортеж в список с помощью встроенной функции `list` и использует метод списка `sort`.

Метод `sort` написан на языке C, поэтому иногда он может обеспечивать более высокую производительность по сравнению с другими версиями функции, но линейный характер сканирования в первых двух версиях в большинстве случаев обеспечивает им более высокую скорость.[1] Файл *mins.py* содержит реализацию всех трех решений:

```
def min1(*args):
    res = args[0]
    for arg in args[1:]:
        if arg < res:
            res = arg
    return res

def min2(first, *rest):
    for arg in rest:
        if arg < first:
            first = arg
    return first
```

[1] В действительности все очень не просто. Функция sort в языке Python написана на языке C и реализует высокоэффективный алгоритм, который пытается использовать существующий порядок следования сортируемых элементов. Этот алгоритм называется «timsort» в честь его создателя Тима Петерса (Tim Peters), а в его документации говорится, что время от времени он показывает «сверхъестественную производительность» (очень неплохую для сортировки!). Однако сортировка по своей природе является экспоненциальной операцией (в процессе сортировки необходимо делить последовательность на составляющие и снова объединять их много раз), тогда как другие версии используют линейные алгоритмы сканирования слева направо. В результате сортировка выполняется быстрее, когда элементы последовательности частично упорядочены, и медленнее в других случаях. Даже при всем при этом производительность самого интерпретатора может изменяться по времени и тот факт, что сортировка реализована на языке C, может существенно помочь. Для более точного анализа производительности вы можете использовать модули `time` и `timeit`, с которыми мы познакомимся в следующей главе.

```
def min3(*args):
    tmp = list(args)     # Или, в Python 2.4+: return sorted(args)[0]
    tmp.sort()
    return tmp[0]

print min1(3,4,1,2)
print min2("bb", "aa")
print min3([2,2], [1,1], [3,3])
```

Все три решения дают одинаковые результаты. Попробуйте несколько раз вызвать функции в интерактивной оболочке, чтобы поэкспериментировать с ними самостоятельно:

```
% python mins.py
1
aa
[1, 1]
```

Обратите внимание, что ни один из этих трех вариантов не выполняет проверку ситуации, когда функции не передается ни одного аргумента. Такую проверку можно было бы предусмотреть, но в этом нет никакой необходимости – во всех трех решениях интерпретатор автоматически возбудит исключение, если та или иная функция не получит ни одного аргумента. В первом случае исключение будет возбуждено при попытке получить нулевой элемент; во втором – когда обнаружится несоответствие списка аргументов; и в третьем – когда функция попытается вернуть нулевой элемент.

Это именно то, что нам нужно, потому что эти функции поддерживают поиск среди данных любого типа и не существует какого-то особого значения, которое можно было бы вернуть в качестве признака ошибки. Из этого правила есть свои исключения (например, когда приходится выполнить дорогостоящие действия, прежде чем появится ошибка), но вообще лучше исходить из предположения, что аргументы не будут вызывать ошибок в работе программного кода функции, и позволить интерпретатору возбуждать исключения, когда этого не происходит.

Дополнительные баллы

Студенты и читатели могут получить дополнительные баллы, если изменят эти функции так, что они будут отыскивать не минимальное, а *максимальное* значение. Сделать это достаточно просто: в первых двух версиях достаточно заменить < на >, а третья версия должна возвращать не элемент `tmp[0]`, а элемент `tmp[-1]`. Дополнительные баллы будут начислены тем, кто догадается изменить имя функции на «max» (хотя это совершенно необязательно).

Кроме того, вполне возможно обобщить функцию так, что она будет отыскивать либо минимальное, либо максимальное значение, определяя отношения элементов за счет интерпретации строки выражения с помощью таких средств, как встроенная функция `eval` (подробности в руководстве к библиотеке), или передавая произвольную функцию

сравнения. В файле *minmax.py* содержится реализация последнего варианта:

```
def minmax(test, *args):
    res = args[0]
    for arg in args[1:]:
        if test(arg, res):
            res = arg
    return res

def lessthan(x, y): return x < y           # См. также: lambda
def grtrthan(x, y): return x > y

print minmax(lessthan, 4, 2, 1, 5, 6, 3)   # Тестирование
print minmax(grtrthan, 4, 2, 1, 5, 6, 3)

% python minmax.py
1
6
```

Функции – это одна из разновидностей объектов, которые могут передаваться в функции, как в этом случае. Например, чтобы заставить функцию отыскивать максимальное (или любое другое) значение, мы могли бы просто передать ей нужную функцию `test`. На первый взгляд может показаться, что мы делаем лишнюю работу, однако главное преимущество такого обобщения функций (вместо того, чтобы содержать две версии, отличающиеся единственным символом) заключается в том, что в будущем нам может потребоваться изменить одну функцию, а не две.

Заключение

Конечно, это было всего лишь упражнение. В действительности нет никаких причин создавать функции `min` и `max`, потому что обе они уже имеются в языке Python! Встроенные версии функций работают практически так же, как и наши, но они написаны на языке C для получения более высокой скорости работы.

Более полезный пример: универсальные функции

Теперь рассмотрим более полезный пример использования специальных режимов сопоставления аргументов. В конце предыдущей главы мы написали функцию, которая возвращала пересечение двух последовательностей (она отбирала элементы, общие для обеих последовательностей). Ниже приводится версия функции, которая возвращает пересечение произвольного числа последовательностей (одной или более), где используется механизм передачи произвольного числа аргументов в форме `*args` для сбора всех передаваемых аргументов в виде коллекции. Все аргументы передаются в тело функции в составе кортежа, поэтому для их обработки можно использовать простой цикл `for`. Ради интереса мы напишем функцию, возвращающую объединение,

которая также принимает произвольное число аргументов и собирает вместе все элементы, имеющиеся в любом из операндов:

```
def intersect(*args):
    res = []
    for x in args[0]:                # Сканировать первую последовательность
        for other in args[1:]:         # Во всех остальных аргументах
            if x not in other: break   # Общий элемент?
        else:                          # Нет: прервать цикл
            res.append(x)              # Да: добавить элемент в конец
    return res

def union(*args):
    res = []
    for seq in args:           # Для всех аргументов
        for x in seq:          # Для всех элементов
            if not x in res:
                res.append(x) # Добавить новый элемент в результат
    return res
```

Поскольку эти функции могут использоваться многократно (и они слишком большие, чтобы вводить их в интерактивной оболочке), мы сохраним их в модуле с именем *inter2.py* (подробнее о модулях рассказывается в пятой части книги). В обе функции аргументы передаются в виде кортежа `args`. Как и оригинальная версия `intersect`, обе они работают с любыми типами последовательностей. Ниже приводится пример обработки строк, последовательностей разных типов и случай обработки более чем двух последовательностей:

```
% python
>>> from inter2 import intersect, union
>>> s1, s2, s3 = "SPAM", "SCAM", "SLAM"

>>> intersect(s1, s2), union(s1, s2)          # Два операнда
(['S', 'A', 'M'], ['S', 'P', 'A', 'M', 'C'])

>>> intersect([1,2,3], (1,4))                 # Смешивание типов
[1]

>>> intersect(s1, s2, s3)                     # Три операнда
['S', 'A', 'M']

>>> union(s1, s2, s3)
['S', 'P', 'A', 'M', 'C', 'L']
```

Следует заметить, что в языке Python появился новый тип данных – множества (описывается в главе 5), поэтому, строго говоря, ни одна из этих функций больше не требуется, – они включены в книгу только для демонстрации подходов к программированию функций. (Так как Python постоянно улучшается, наблюдается странная тенденция – мои книжные примеры делаются устаревшими с течением времени!)

Сопоставление аргументов: практические детали

Ниже приводится несколько правил в языке Python, которым вам необходимо следовать, если у вас появится потребность использовать специальные режимы сопоставления аргументов:

- В вызове функции все аргументы, которые передаются не по ключу (`name`), должны находиться в начале списка, далее должны следовать аргументы, которые передаются по ключу (`name=value`), вслед за ними должна следовать форма `*name` и, наконец, `**name`.
- В заголовке функции аргументы должны стоять в том же самом порядке: обычные аргументы (`name`), вслед за ними аргументы со значениями по умолчанию (`name=value`), далее форма `*name`, если необходимо, и, наконец, форма `**name`.

Если поставить аргументы в любом другом порядке, вы получите сообщение о синтаксической ошибке, потому что в этих случаях комбинация получается неоднозначной. Сам интерпретатор использует следующий порядок сопоставления аргументов перед выполнением операций присваивания:

1. Определяются позиционные аргументы, которые передаются не по ключу.
2. Определяются аргументы, которые передаются по ключу, и производится сопоставление имен.
3. Определяются и помещаются в кортеж `*name` дополнительные аргументы, которые передаются не по ключу.
4. Определяются и помещаются в словарь `**name` дополнительные аргументы, которые передаются по ключу.
5. Определяются аргументы, имеющие значения по умолчанию, которые отсутствуют в вызове.

После этого интерпретатор убеждается, что в каждом аргументе передается единственное значение, в противном возбуждает исключение. Это действительно так же сложно, как выглядит, но изучение алгоритма сопоставления аргументов в языке Python поможет вам разобраться с некоторыми сложными случаями, особенно когда одновременно используются несколько режимов. Дополнительные примеры со специальными режимами сопоставления приводятся в упражнениях в конце четвертой части.

Как видите, режимы сопоставления аргументов могут быть весьма сложными. Но использовать их совершенно необязательно – вы можете ограничиться исключительно позиционными аргументами и это, пожалуй, будет наилучший выбор. Однако некоторые инструменты языка Python используют эти режимы, поэтому их понимание играет важную роль.

Придется держать в уме: аргументы, передаваемые по ключу

Аргументы, которые передаются по ключу, играют важную роль в библиотеке Tkinter, которая фактически стала стандартным средством для разработки графического интерфейса в языке Python. Мы познакомимся с Tkinter далее в этой книге, но в качестве предварительного знакомства замечу, что при использовании этой библиотеки для установки значений параметров компонентов графического интерфейса используются аргументы, передаваемые по ключу. Например, следующий вызов:

```
from Tkinter import *
widget = Button(text="Press me", command=someFunction)
```

создает новую кнопку и определяет текст на кнопке и функцию обратного вызова, с помощью ключевых аргументов `text` и `command`. Так как графические компоненты могут иметь большое число параметров, аргументы, передаваемые по ключу, позволяют указывать только необходимые вам параметры. В противном случае пришлось бы перечислять все возможные параметры в соответствии с их позициями или надеяться, что аргументы со значениями по умолчанию будут правильно интерпретироваться во всех возможных ситуациях.

В заключение

В этой главе мы изучили две ключевые концепции, имеющие отношение к функциям: области видимости (как выполняется поиск переменных при обращениях к ним) и аргументы (как объекты передаются в функции). Здесь мы узнали, что переменные считаются локальными для определений функций, где выполняется присваивание значений этим переменным при условии, что они не объявлены как глобальные. Кроме того, мы узнали, что аргументы передаются функции через операцию присваивания, т. е. в виде ссылок на объекты, которые в действительности являются указателями.

Мы также познакомились с дополнительными особенностями областей видимости и аргументов, например с областями видимости вложенных функций и аргументами, которые передаются по ключу. Наконец мы познакомились с некоторыми рекомендациями по проектированию приложений (стремиться минимизировать количество глобальных переменных и избегать изменений переменных в соседних файлах) и увидели, что изменяемые объекты в аргументах проявляют то же самое поведение, как и другие разделяемые ссылки на объекты, – если функции явно не передается копия объекта, воздействие непо-

средственно на изменяемый объект может отразиться на вызывающей программе.

Следующая глава завершает тему функций исследованием более сложных концепций, связанных с функциями: `lambda`-выражений, генераторов, итераторов, функциональных инструментов, таких как `map`, и т. д. Многие из этих концепций исходят из того, что функции в языке Python являются обычными объектами и потому поддерживают дополнительные, очень гибкие режимы работы. Однако, прежде чем углубиться в эти темы, изучите контрольные вопросы к этой главе, чтобы закрепить знания, полученные здесь.

Закрепление пройденного

Контрольные вопросы

1. Что выведет следующий фрагмент и почему?

```
>>> X = 'Spam'
>>> def func():
...     print X
...
>>> func()
```

2. Что выведет следующий фрагмент и почему?

```
>>> X = 'Spam'
>>> def func():
...     X = 'NI!'
...
>>> func()
>>> print X
```

3. Что выведет следующий фрагмент и почему?

```
>>> X = 'Spam'
>>> def func():
...     X = 'NI'
...     print X
...
>>> func()
>>> print X
```

4. Что выведет следующий фрагмент на этот раз и почему?

```
>>> X = 'Spam'
>>> def func():
...     global X
...     X = 'NI'
...
>>> func()
>>> print X
```

5. Что можно сказать об этом фрагменте – что он выведет и почему?

```
>>> X = 'Spam'
>>> def func():
...     X = 'NI'
...     def nested():
...         print X
...     nested()
...
>>> func()
>>> X
```

6. И в последний раз: что выведет следующий фрагмент и почему?

```
>>> def func(a, b, c=3, d=4): print a, b, c, d
...
>>> func(1, *(5,6))
```

7. Назовите три-четыре способа в языке Python сохранять информацию о состоянии в функциях.
8. Назовите три способа, которые могут использоваться для передачи результатов из функции в вызывающую программу.

Ответы

1. В данном случае будет выведена строка `'Spam'`, потому что функция обращается к глобальной переменной в объемлющем модуле (если внутри функции переменной не присваивается значение, она интерпретируется как глобальная).
2. В данном случае снова будет выведена строка `'Spam'`, потому что операция присваивания внутри функции создает локальную переменную и тем самым скрывает глобальную переменную с тем же именем. Инструкция `print` находит неизмененную переменную в глобальной области видимости.
3. Будет выведена последовательность символов `'Ni'` в одной строке и `'Spam'` – в другой, потому что внутри функции инструкция `print` найдет локальную переменную, а за ее пределами – глобальную.
4. На этот раз будет выведена строка `'Ni'`, потому что объявление `global` предписывает выполнять присваивание внутри функции переменной, находящейся в глобальной области видимости объемлющего модуля.
5. В этом случае снова будет выведена последовательность символов `'Ni'` в одной строке и `'Spam'` – в другой, потому что инструкция `print` во вложенной функции отыщет имя в локальной области видимости объемлющей функции, а инструкция `print` в конце фрагмента отыщет имя в глобальной области видимости.
6. Здесь будет выведено `"1 5 6 4"`: `1` соответствует аргументу в первой позиции, `5` и `6` соответствуют аргументам `b` и `c` в соответствии с фор-

мой `*name` (значение `6` переопределяет значение по умолчанию аргумента `c`) и `d` получит значение по умолчанию `4`, потому что четвертый аргумент в вызове функции отсутствует.

7. Так как значения локальных переменных исчезают, когда функция возвращает управление, то информацию о состоянии в языке Python можно сохранять в глобальных переменных, для вложенных функций – в области видимости объемлющих функций, а также посредством аргументов со значениями по умолчанию. Альтернативный способ заключается в использовании классов и приемов ООП, который обеспечивает лучшую поддержку возможности сохранения информации о состоянии, чем любой из трех предыдущих приемов, потому что этот способ делает сохранение явным, позволяя выполнять присваивание значений атрибутам.
8. Функции могут возвращать результаты с помощью инструкции `return`, воздействуя на изменяемые объекты, передаваемые в аргументах, а также изменением глобальных переменных. Вообще глобальные переменные использовать для этих целей не рекомендуется (за исключением редких случаев, таких как многопоточные программы), потому что это осложняет понимание и использование программного кода. Наилучшим способом является инструкция `return`, хотя воздействие на изменяемые объекты – тоже неплохой вариант при условии, что он предусмотрен заранее. Кроме того, функции могут выполнять обмен информацией через такие системные устройства, как файлы и сокеты, но это уже выходит за рамки данной главы.