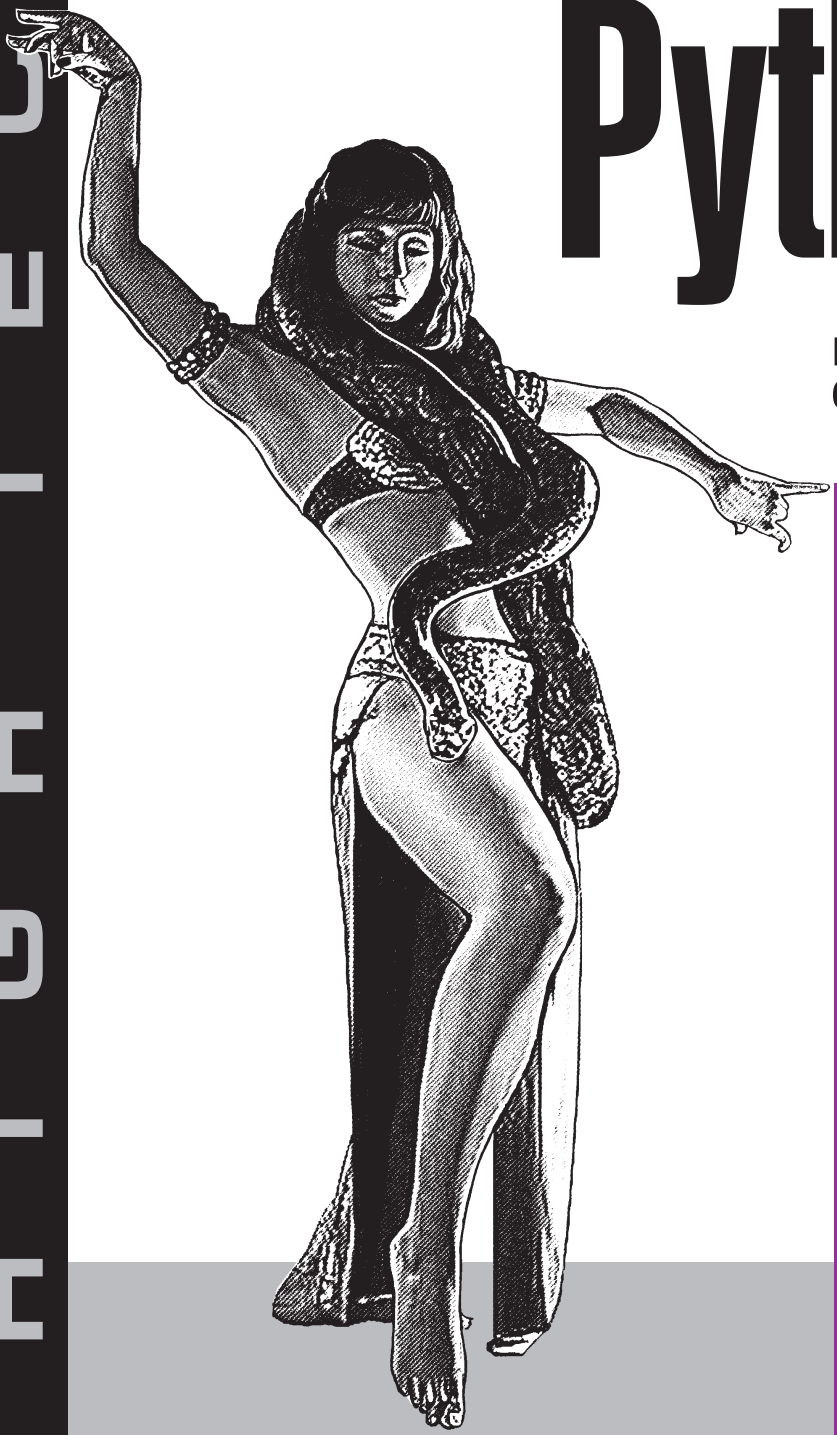
HIGH TECH

ДЭВИД БИЗЛИ

четвертое издание

Python

ПОДРОБНЫЙ СПРАВОЧНИК



Символ®

Python

Essential Reference

Fourth Edition

David Beazley

Addison-Wesley

HIGH TECH

Python

Подробный справочник

Четвертое издание

Дэвид Бизли

Символ®

Санкт-Петербург — Москва
2010

Серия «High tech»

Дэвид Бизли

Python. Подробный справочник

Перевод А. Киселева

Главный редактор	*А. Галунов*
Зав. редакцией	*Н. Макарова*
Выпускающий редактор	*П. Щеголев*
Редактор	*Ю. Бочина*
Корректор	*С. Минин*
Верстка	*К. Чубаров*

Бизли Д.

Python. Подробный справочник. – Пер. с англ. – СПб.: Символ-Плюс, 2010. – 864 с., ил.

ISBN 978-5-93286-157-8

«Python. Подробный справочник» – это авторитетное руководство и детальный путеводитель по языку программирования Python. Книга предназначена для практикующих программистов; она компактна, нацелена на суть дела и написана очень доступным языком. Она детально описывает не только ядро языка, но и наиболее важные части стандартной библиотеки Python. Дополнительно освещается ряд тем, которые не рассматриваются ни в официальной документации, ни в каких-либо других источниках.

Читателю предлагается практическое знакомство с особенностями Python, включая генераторы, сопрограммы, замыкания, метаклассы и декораторы. Подробно описаны новые модули, имеющие отношение к разработке многозадачных программ, использующих потоки управления и дочерние процессы, а также предназначенные для работы с системными службами и организации сетевых взаимодействий.

В полностью переработанном и обновленном четвертом издании улучшена организация материала, что позволяет еще быстрее находить ответы на вопросы и обеспечивает еще большее удобство работы со справочником. Книга отражает наиболее существенные нововведения в языке и в стандартной библиотеке, появившиеся в Python 2.6 и Python 3.

ISBN 978-5-93286-157-8
ISBN 978-0-672-32978-4 (англ)

Издательство «Символ-Плюс». 199034, Санкт-Петербург, 16 линия, 7,
тел. (812) 324-5353, www.symbol.ru. Лицензия ЛП N 000054 от 25.12.98.

Подписано в печать 30.07.2010. Формат 70×100 $^{1}/_{16}$. Печать офсетная.
Объем 54 печ. л. Тираж 1200 экз. Заказ №
Отпечатано с готовых диапозитивов в ГУП «Типография «Наука»
199034, Санкт-Петербург, 9 линия, 12.

Посвящается Пауле, Томасу
и его будущему брату.

Оглавление

Об авторе

Дэвид М. Бизли (David M. Beazley) является давним приверженцем Python, примкнувшим к сообществу разработчиков этого языка еще в 1996 году. Наибольшую известность, пожалуй, он получил за свою работу над популярным программным пакетом SWIG, позволяющим использовать программы и библиотеки, написанные на C/C++, в других языках программирования, включая Python, Perl, Ruby, Tcl и Java. Он также является автором множества других программных инструментов, включая PLY, реализацию инструментов lex[1] и yacc[2] на языке Python. В течение семи лет Дэвид работал в Отделении теоретической физики Лос-Аламосской Национальной лаборатории и возглавлял работы по интеграции Python с высокопроизводительным программным обеспечением моделирования процессов для параллельных вычислительных систем. После этого он зарабатывал славу злого профессора, обожая озадачивать студентов сложными проектами в области программирования. Однако затем он понял, что это не совсем его дело, и теперь живет в Чикаго и работает как независимый программист, консультант, преподаватель по языку Python и иногда как джазовый музыкант. Связаться с ним можно по адресу *http://www.dabeaz.com*.

---

[1] Генератор лексических анализаторов. – *Прим. перев.*

[2] Генератор синтаксических анализаторов. – *Прим. перев.*

О научном редакторе

Ноа Гифт (Noah Gift) – соавтор книги «Python For UNIX and Linux System Administration» (издательство O'Reilly).[1] Он также принимал участие в работе над книгой «Google App Engine In Action» (издательство Manning). Гифт является автором, лектором, консультантом и ведущим специалистом. Пишет статьи для таких изданий, как «IBM developerWorks», «Red Hat Magazine», сотрудничает с издательствами O'Reilly и MacTech. Домашняя страница его консалтинговой компании находится по адресу *http://www.giftcs.com*, а кроме того, значительное количество его статей можно найти по адресу *http://noahgift.com*. Помимо этого, вы можете стать последователем Ноа на сайте микроблогинга Twitter.

Ноа получил степень магистра по специальности «Теория информационных и вычислительных систем» в Калифорнийском университете, в городе Лос-Анджелес, степень бакалавра диетологии – в Калифорнийском государственном политехническом университете, в городе Сан Луис Обиспо. Имеет сертификаты системного администратора от компании Apple и LPI. Работал в таких компаниях, как Caltech, Disney Feature Animation, Sony Imageworks и Turner Studios. В настоящее время он работает в компании Weta Digital, в Новой Зеландии. В свободное время любит гулять со своей женой Леа и сыном Лиамом, играть на пианино, бегать на длинные дистанции и регулярно тренируется.

---

[1] Н. Гифт, Дж. Джонс «Python в системном администрировании UNIX и Linux». – Пер. с англ. – СПб.: Символ-Плюс, 2009.

Благодарности

Эта книга не смогла бы увидеть свет без поддержки многих людей. Прежде всего, я хотел бы поблагодарить Ноа Гифта, который включился в проект и оказал существенную помощь в подготовке четвертого издания. Курт Грандис (Kurt Grandis) дал множество полезных комментариев ко многим главам. Мне также хотелось бы поблагодарить бывших технических редакторов Тимоти Борончика (Timothy Boronczyk), Пола Дюбойса (Paul DuBois), Мэтса Викманна (Mats Wichmann), Дэвида Ашера (David Ascher) и Тима Белла (Tim Bell) за их ценные комментарии и советы, принесшие успех предыдущим изданиям. Гвидо ван Россум (Guido van Rossum), Джереми Хилтон (Jeremy Hylton), Фред Дрейк (Fred Drake), Роджер Масс (Roger Masse) и Барри Варшав (Barry Warsaw) оказали весьма существенную помощь в подготовке первого издания, пока гостили у меня несколько недель жарким летом 1999 года. Наконец, эта книга едва ли была бы возможна без отзывов, которые я получал от читателей. Их слишком много, чтобы я мог перечислить их поименно, тем не менее я приложил все усилия, чтобы учесть все поступившие предложения по улучшению книги. Я также хотел бы поблагодарить сотрудников издательств Addison-Wesley и Pearson Education за их непрекращающуюся помощь проекту. Марк Табер (Mark Taber), Майкл Торстон (Michael Thurston), Сет Керни (Seth Kerney) и Лайза Тибо (Lisa Thibault) сделали все возможное, чтобы обеспечить выход этого издания. Отдельное спасибо Робину Дрейку (Robin Drake), невероятные усилия которого обеспечили появление третьего издания. Наконец, я хотел бы поблагодарить мою замечательную жену и друга Паулу Камен за ее поддержку, потрясающий юмор и любовь.

Нам интересны ваши отзывы

Вы, будучи читателями этой книги, являетесь наиболее важными критиками. Ваше мнение ценно для нас, и нам хотелось бы знать, что было сделано правильно, что можно было бы сделать лучше, книги на какие темы вы хотели бы увидеть и любые другие ваши пожелания.

Вы можете отправлять свои сообщения по электронной почте или написать прямо мне о том, что вам не понравилось в этой книге и что мы могли бы сделать, чтобы сделать книгу еще лучше.

Учтите, что я не в состоянии помочь вам в решении технических проблем, имеющих отношение к теме этой книги, и что из-за большого объема сообщений, поступающих мне по электронной почте, я не могу ответить на каждое из них.

Не забудьте включить в текст своего письма название этой книги и имя автора, а также ваше имя, номер телефона или адрес электронной почты. Я внимательно ознакомлюсь со всеми вашими комментариями и поделюсь ими с автором и редакторами, работавшими над книгой.

Эл. почта: feedback@developers-library.info

Адрес: Mark Taber
Associate Publisher
Pearson Education
800 East 96th Street
Indianapolis, IN 46240 USA

Поддержка читателей

Посетите наш веб-сайт и зарегистрируйте свою копию книги по адресу *informit.com/register*, чтобы получить доступ к последним дополнениям, загружаемым архивам с примерами и списку обнаруженных опечаток.

Введение

Эта книга задумывалась как краткий справочник по языку программирования Python. Несмотря на то что опытный программист будет в состоянии изучить Python с помощью этой книги, тем не менее она не предназначена для использования в качестве учебника по программированию. Основная ее цель состоит в том, чтобы максимально точно и кратко описать ядро языка Python и наиболее важные части библиотеки Python. Эта книга предполагает, что читатель уже знаком с языком Python или имеет опыт программирования на таких языках, как C или Java. Кроме того, общее знакомство с принципами системного программирования (например, с основными понятиями операционных систем и с разработкой сетевых приложений) может быть полезным для понимания описания некоторых частей библиотеки.

Вы можете бесплатно загрузить Python на сайте *http://www.python.org*. Здесь вы найдете версии практически для всех операционных систем, включая UNIX, Windows и Macintosh. Кроме того, на веб-сайте Python вы найдете ссылки на документацию, практические руководства и разнообразные пакеты программного обеспечения сторонних производителей.

Появление данного издания книги «Python. Подробный справочник» совпало с практически одновременным выходом версий Python 2.6 и Python 3.0. В версии Python 3 была нарушена обратная совместимость с предшествующими версиями языка. Как автор и программист я столкнулся с дилеммой: просто перейти к версии Python 3.0 или продолжать опираться на версии Python 2.x, более знакомые большинству программистов?

Несколько лет тому назад, будучи программистом на языке C, я взял себе за правило воспринимать определенные книги как окончательное руководство по использованию особенностей языка программирования. Например, если вы используете какие-то особенности, не описанные в книге Кернигана и Ритчи, скорее всего, они будут непереносимы и их следует использовать с осторожностью. Этот подход сослужил мне неплохую службу как программисту, и именно его я решил использовать в данном издании «Подробного справочника». В частности, я предпочел опустить особенности версии Python 2, которые были удалены из версии Python 3. Точно так же я не буду останавливаться на особенностях версии Python 3, которые нарушают обратную совместимость (впрочем, эти особенности я описал в отдельном приложении). На мой взгляд, в результате получилась книга, которая будет полезна программистам на языке Python независимо от используемой ими версии.

Четвертое издание книги «Python. Подробный справочник» также включает важные дополнения по сравнению с первым изданием, вышедшим десять лет тому назад.[1] Развитие языка Python в течение последних лет в основном продолжалось в направлении добавления новых особенностей; особенно это относится к функциональному и метапрограммированию. В результате главы, посвященные функциональному и объектно-ориентированному программированию, были существенно расширены и теперь охватывают такие темы, как генераторы, итераторы, сопрограммы, декораторы и метаклассы. В главы, посвященные стандартной библиотеке Python, были добавлены описания наиболее современных модулей. Кроме того, во всей книге были обновлены исходные тексты примеров. Я полагаю, что большинство программистов с радостью встретят дополненное описание.

Наконец, следует отметить, что **Python включает тысячи страниц документации**. Содержимое этой книги в значительной степени основано на этой документации, однако здесь имеется множество важных отличий. Во-первых, информация в этом справочнике представлена в более компактной форме и содержит различные примеры и дополнительные описания многих тем. Во-вторых, описание библиотеки языка Python было значительно расширено и включает дополнительный справочный материал. В особенности это относится к описаниям низкоуровневых системных и сетевых модулей, эффективное использование которых сопряжено с применением несметного количества параметров, перечисленных в руководствах и справочниках. Помимо этого, с целью достижения большей краткости были опущены описания ряда устаревших и не рекомендуемых к использованию библиотечных модулей.

Приступая к работе над этой книгой, главной моей целью было создание справочника, содержащего информацию практически обо всех аспектах использования языка Python и его огромной коллекции модулей. Несмотря на то что эту книгу нельзя назвать мягким введением в язык программирования **Python, я надеюсь, что она сможет послужить полезным дополнением** к другим книгам в течение нескольких лет. Я с благодарностью приму любые ваши комментарии.

Дэвид Бизли (David Beazley)

Чикаго, Иллинойс
Июнь, 2009

[1] Д. Бизли «Язык программирования Python. Справочник». – Пер. с англ. – Киев: ДиаСофт, 2000. – *Прим. перев.*

I

Язык программирования Python

1

Вводное руководство

Данная глава представляет собой краткое введение в язык программирования Python. Цель ее состоит в том, чтобы продемонстрировать наиболее важные особенности языка Python, не погружаясь при этом в описание деталей. Следуя поставленной цели, эта глава кратко описывает основные понятия языка, такие как переменные, выражения, конструкции управления потоком выполнения, функции, генераторы, классы, а также ввод-вывод. Эта глава не претендует на полноту охвата рассматриваемых тем. Однако опытные программисты без труда смогут на основе представленных здесь сведений создавать достаточно сложные программы. Начинающие программисты могут опробовать некоторые примеры, чтобы получить представление об особенностях языка. Если вы только начинаете изучать язык Python и используете версию Python 3, эту главу лучше будет изучать, опираясь на версию Python 2.6. Практически все основные понятия в равной степени применимы к обеим версиям, однако в Python 3 появилось несколько важных изменений в синтаксисе, в основном связанных с вводом и выводом, которые могут отрицательно сказаться на работоспособности многих примеров в этом разделе. За дополнительной информацией обращайтесь к приложению «Python 3».

Вызов интерпретатора

Программы на языке Python выполняются интерпретатором. Обычно интерпретатор вызывается простым вводом команды `python`. Однако существует множество различных реализаций интерпретатора и разнообразных сред разработки (например, Jython, IronPython, IDLE, ActivePython, Wing IDE, pydev и так далее), поэтому за информацией о порядке вызова следует обращаться к документации. После запуска интерпретатора в командной оболочке появляется приглашение к вводу, в котором можно вводить программы, исполняемые в простом цикле чтение-выполнение. Например, ниже представлен фрагмент вывода, где интерпретатор отображает сообщение с упоминанием об авторских правах и выводит приглашение к вводу >>>, где пользователь ввел знакомую команду «Привет, Мир»:

```
Python 2.6rc2 (r26rc2:66504, Sep 19 2008, 08:50:24)
[GCC 4.0.1 (Apple Inc. build 5465)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> print "Привет, Мир"
Привет, Мир
>>>
```

Примечание

Если при попытке опробовать предыдущий пример была получена ошибка SyntaxError, это может свидетельствовать о том, что вы пользуетесь версией Python 3. В этом случае вы по-прежнему можете следовать за примерами этой главы, но при этом должны помнить, что инструкция print в версии Python 3 была преобразована в функцию. Чтобы избавиться от ошибки, просто заключите выводимые значения в круглые скобки. Например:

```
>>> print("Привет, Мир")
Привет, Мир
>>>
```

Добавление круглых скобок допускается и при использовании Python 2, при условии, что выводится единственный элемент. Однако такой синтаксис редко можно встретить в существующем программном коде на языке Python. В последующих главах этот синтаксис иногда будет использоваться в примерах, основная цель которых не связана с выводом информации, но которые, как предполагается, должны одинаково работать при использовании любой из версий Python 2 или Python 3.

Интерактивный режим работы интерпретатора Python является одной из наиболее полезных особенностей. В интерактивной оболочке можно вводить любые допустимые инструкции или их последовательности и тут же получать результаты. Многие, включая и автора, используют интерпретатор Python в интерактивном режиме в качестве настольного калькулятора. Например:

```
>>> 6000 + 4523.50 + 134.12
10657.620000000001
>>> _ + 8192.32
18849.940000000002
>>>
```

При использовании интерпретатора Python в интерактивном режиме можно использовать специальную переменную _, которая хранит результат последней операции. Ее можно использовать для хранения промежуточных результатов при выполнении последовательности инструкций. Однако важно помнить, что эта переменная определена только при работе интерпретатора в интерактивном режиме.

Если вам необходимо написать программу, которая будет использоваться не один раз, поместите инструкции в файл, как показано ниже:

```
# helloworld.py
print "Привет, Мир"
```

Файлы с исходными текстами программ на языке Python являются обычными текстовыми файлами и обычно имеют расширение .py. Символ # отмечает начало комментария, простирающегося до конца строки.

Чтобы выполнить программу в файле helloworld.py, необходимо передать имя файла интерпретатору, как показано ниже:

```
% python helloworld.py
Привет, Мир
%
```

Чтобы запустить программу на языке Python в Windows, можно дважды щелкнуть мышью на файле с расширением .py или ввести имя программы в окне Запустить... (Run command), которое вызывается одноименным пунктом в меню Пуск (Start). В результате в окне консоли будет запущен интерпретатор, который выполнит указанную программу. Однако следует помнить, что окно консоли будет закрыто сразу же после окончания работы программы (зачастую еще до того, как вы успеете прочитать результаты). При отладке лучше всего запускать программы с помощью инструментов разработчика на языке Python, таких как IDLE.

В UNIX можно добавить в первую строку программы последовательность #!, как показано ниже:

```
#!/usr/bin/env python
print "Привет, Мир"
```

Интерпретатор последовательно выполняет инструкции, пока не достигнет конца файла. При работе в интерактивном режиме завершить сеанс работы с интерпретатором можно вводом символа EOF (end of file – конец файла) или выбором пункта меню Exit (Выйти) в среде разработки Python. В UNIX символ EOF вводится комбинацией клавиш Ctrl+D; в Windows – Ctrl+Z. Программа может совершить выход, возбудив исключение SystemExit.

```
>>> raise SystemExit
```

Переменные и арифметические выражения

В листинге 1.1 приводится программа, которая использует переменные и выражения для вычисления сложных процентов.

Листинг 1.1. Простое вычисление сложных процентов

```
principal = 1000     # Начальная сумма вклада
rate = 0.05          # Процент
numyears = 5         # Количество лет
year = 1
while year <= numyears:
    principal = principal * (1 + rate)
    print year, principal     # В Python 3: print(year, principal)
    year += 1
```

В результате работы программы будет получена следующая таблица:

```
1 1050.0
2 1102.5
3 1157.625
4 1215.50625
5 1276.2815625
```

Python – это язык с динамической типизацией, то есть в ходе выполнения программы одна и та же переменная может хранить значения различных типов. Оператор присваивания просто создает связь между именем переменной и значением. Хотя каждое значение имеет собственный тип данных, например целое число или строка, сами переменные не имеют типа и в процессе выполнения программы могут ссылаться на значения любых типов. Этим Python отличается от языка C, например, в котором каждая переменная имеет определенный тип, размер и местоположение в памяти, где сохраняется ее значение. Динамическую природу языка Python можно наблюдать в листинге 1.1 на примере переменной `principal`. Изначально ей присваивается целочисленное значение. Однако позднее в программе выполняется следующее присваивание:

```
principal = principal * (1 + rate)
```

Эта инструкция вычисляет выражение и присваивает результат переменной с именем `principal`. Несмотря на то что первоначальное значение переменной `principal` было целым числом `1000`, новое значение является числом с плавающей точкой (значение переменной `rate` является числом с плавающей точкой, поэтому результатом приведенного выше выражения также будет число с плавающей точкой). То есть в середине программы «тип» переменной `principal` динамически изменяется с целочисленного на число с плавающей точкой. Однако, если быть более точными, следует заметить, что изменяется не тип переменной `principal`, а тип значения, на которое ссылается эта переменная.

Конец строки завершает инструкцию. Однако имеется возможность разместить несколько инструкций в одной строке, отделив их точкой с запятой, как показано ниже:

```
principal = 1000; rate = 0.05; numyears = 5;
```

Инструкция `while` вычисляет условное выражение, следующее прямо за ней. Если результат выражения оценивается как истина, выполняется тело инструкции `while`. Условное выражение, а вместе с ним и тело цикла, вычисляется снова и снова, пока не будет получено ложное значение. Тело цикла выделено отступами, то есть в листинге 1.1 на каждой итерации выполняются три инструкции, следующие за инструкцией `while`. Язык Python не предъявляет жестких требований к величине отступов, важно лишь, чтобы в пределах одного блока использовались отступы одного и того же размера. Однако чаще всего отступы оформляются четырьмя пробелами (и так и рекомендуется).

Один из недостатков программы, представленной в листинге 1.1, заключается в отсутствии форматирования при выводе данных. Чтобы исправить

его, можно было бы использовать выравнивание значений переменной principal по правому краю и ограничить точность их представления двумя знаками после запятой. Добиться такого форматирования можно несколькими способами. Наиболее часто для подобных целей используется оператор форматирования строк (%), как показано ниже:

```
print "%3d %0.2f" % (year, principal)
print("%3d %0.2f" % (year, principal)) # Python 3
```

Теперь вывод программы будет выглядеть, как показано ниже:

```
1 1050.00
2 1102.50
3 1157.63
4 1215.51
5 1276.28
```

Строки формата содержат обычный текст и специальные спецификаторы формата, такие как "%d", "%s" и "%f". Приведенные спецификаторы определяют формат представления данных определенных типов – целых чисел, строк и чисел с плавающей точкой соответственно. Спецификаторы формата могут также содержать модификаторы, определяющие ширину поля вывода и точность представления значений. Например, спецификатор "%3d" форматирует целое число, с выравниванием по правому краю в поле шириной 3 символа, а спецификатор "%0.2f" форматирует число с плавающей точкой так, чтобы выводились только два знака после запятой. Поведение спецификаторов формата для строк во многом идентично поведению спецификаторов формата для функции printf() в языке C и подробно описывается в главе 4 «Операторы и выражения».

Более современный подход к форматированию строк заключается в форматировании каждой части строки по отдельности, с помощью функции format(). Например:

```
print format(year,"3d"),format(principal,"0.2f")
print(format(year,"3d"),format(principal,"0.2f")) # Python 3
```

Спецификаторы формата для функции format() похожи на спецификаторы, традиционно используемые в операторе форматирования строк (%). Например, спецификатор "3d" форматирует целое число, выравнивая его по правому краю в поле шириной 3 символа, а спецификатор "0.2f" форматирует число с плавающей точкой, ограничивая точность представления двумя знаками после запятой. Кроме того, строки имеют собственный метод format(), который может использоваться для форматирования сразу нескольких значений. Например:

```
print "{0:3d} {1:0.2f}".format(year,principal)
print("{0:3d} {1:0.2f}".format(year,principal)) # Python 3
```

В данном примере число перед двоеточием в строках "{0:3d}" и "{1:0.2f}" определяет порядковый номер аргумента метода format(), а часть после двоеточия – спецификатор формата.

Условные операторы

Для простых проверок можно использовать инструкции `if` и `else`. Например:

```
if a < b:
    print “Компьютер говорит Да”
else:
    print “Компьютер говорит Нет”
```

Тела инструкций `if` и `else` отделяются отступами. Инструкция `else` является необязательной.

Чтобы создать пустое тело, не выполняющее никаких действий, можно использовать инструкцию `pass`, как показано ниже:

```
if a < b:
    pass # Не выполняет никаких действий
else:
    print “Компьютер говорит Нет”
```

Имеется возможность формировать булевы выражения с использованием ключевых слов `or`, `and` и `not`:

```
if product == “игра” and type == “про пиратов” \
                    and not (age < 4 or age > 8):
    print “Я беру это!”
```

Примечание

При выполнении сложных проверок строка программного кода может оказаться достаточно длинной. Чтобы повысить удобочитаемость, инструкцию можно перенести на следующую строку, добавив символ обратного слэша (\) в конце строки, как показано выше. В этом случае обычные правила оформления отступов к следующей строке не применяются, поэтому вы можете форматировать строки, продолжающие инструкцию, как угодно.

В языке Python отсутствует специальная инструкция проверки значений, такая как `switch` или `case`. Чтобы выполнить проверку на соответствие нескольким значениям, можно использовать инструкцию `elif`, например:

```
if suffix == “.htm”:
    content = “text/html”
elif suffix == “.jpg”:
    content = “image/jpeg”
elif suffix == “.png”:
    content = “image/png”
else:
    raise RuntimeError(“Содержимое неизвестного типа”)
```

Для определения истинности используются значения `True` и `False` типа Boolean. Например:

```
if ‘spam’ in s:
    has_spam = True
```

```
else:
    has_spam = False
```

Все операторы отношений, такие как < и >, возвращают `True` или `False`. Оператор `in`, задействованный в предыдущем примере, часто используется для проверки вхождения некоторого значения в другой объект, такой как строка, список или словарь. Он также возвращает значение `True` или `False`, благодаря чему предыдущий пример можно упростить, как показано ниже:

```
has_spam = 'spam' in s
```

Операции ввода-вывода с файлами

Следующая программа открывает файл и читает его содержимое построчно:

```
f = open("foo.txt")      # Возвращает файловый объект
line = f.readline()      # Вызывается метод readline() файла
while line:
    print line,          # Завершающий символ ',' предотвращает перевод строки
    # print(line,end='') # Для Python 3
    line = f.readline()
f.close()
```

Функция `open()` возвращает новый файловый объект. Вызывая методы этого объекта, можно выполнять различные операции над файлом. Метод `readline()` читает одну строку из файла, включая завершающий символ перевода строки. По достижении конца файла возвращается пустая строка.

В данном примере программа просто выполняет обход всех строк в файле `foo.txt`. Такой прием, когда программа в цикле выполняет обход некоторой коллекции данных (например, строк в файле, чисел, строковых значений и так далее), часто называют итерациями. Поскольку потребность в итерациях возникает достаточно часто, для этих целей в языке Python предусмотрена специальная инструкция `for`, которая выполняет обход элементов коллекции. Например, та же программа может быть записана более кратко:

```
for line in open("foo.txt"):
    print line,
```

Чтобы записать вывод программы в файл, в инструкцию `print` можно добавить оператор >> перенаправления в файл, как показано в следующем примере:

```
f = open("out","w") # Открывает файл для записи
while year <= numyears:
    principal = principal * (1 + rate)
    print >>f,"%3d %0.2f" % (year,principal)
    year += 1
f.close()
```

Оператор >> можно использовать только в Python 2. При работе с версией Python 3 инструкцию `print` следует изменить, как показано ниже:

```
print("%3d %0.2f" % (year,principal),file=f)
```

Кроме того, файловые объекты обладают методом `write()`, который можно использовать для записи неформатированных данных. Например, инструкцию `print` в предыдущем примере можно было бы заменить следующей инструкцией:

```
f.write("%3d %0.2f\n" % (year,principal))
```

Хотя в этих примерах выполняются операции над файлами, те же приемы можно использовать при работе со стандартными потоками ввода и вывода. Например, когда требуется прочитать ввод пользователя в интерактивном режиме, его можно получить из файла `sys.stdin`. Когда необходимо вывести данные на экран, можно записать их в файл `sys.stdout`, который используется инструкцией `print`. Например:

```
import sys
sys.stdout.write("Введите свое имя :")
name = sys.stdin.readline()
```

При использовании Python 2 этот пример можно сократить еще больше, как показано ниже:

```
name = raw_input("Введите свое имя :")
```

В Python 3 функция `raw_input()` стала называться `input()`, но она действует точно так же.

Строки

Чтобы создать литерал строки, ее необходимо заключить в апострофы, в кавычки или в тройные кавычки, как показано ниже:

```
a = "Привет, Мир!"
b = 'Python - прелесть'
c = """Компьютер говорит 'Нет'"""
```

Строковый литерал должен завершаться кавычками того же типа, как использовались в начале. В противоположность литералам в апострофах и в кавычках, которые должны располагаться в одной логической строке, литералы в тройных кавычках могут включать текст произвольной длины – до завершающих тройных кавычек. Литералы в тройных кавычках удобно использовать, когда содержимое литерала располагается в нескольких строках, как показано ниже:

```
print '''Content-type: text/html

<h1> Hello World </h1>
Click <a href="http://www.python.org">here</a>.
'''
```

Строки хранятся как последовательности символов, доступ к которым можно получить с помощью целочисленного индекса, начиная с нуля. Чтобы извлечь из строки единственный символ, можно использовать оператор индексирования `s[i]`, например:

```
a = “Привет, Мир”
b = a[4] # b = ‘e’
```

Чтобы извлечь подстроку, можно использовать оператор сечения `s[i:j]`. Он извлечет из строки `s` все символы с порядковыми номерами *k* в диапазоне *i* <= *k* < *j*. Если какой-либо из индексов опущен, предполагается, что он соответствует началу или концу строки соответственно:

```
c = a[:6]  # c = “Привет”
d = a[8:]  # d = “Мир”
e = a[3:9] # e = “вет, М”
```

Конкатенация строк выполняется с помощью оператора сложения (+):

```
g = a + “ Это проверка”
```

Интерпретатор Python никогда не интерпретирует строку как число, даже если она содержит только цифровые символы (как это делается в других языках, таких как Perl или PHP). Например, оператор + всегда выполняет конкатенацию строк:

```
x = “37”
y = “42”
z = x + y # z = “3742” (конкатенация строк)
```

Чтобы выполнить арифметическую операцию над значениями, хранящимися в виде строк, необходимо сначала преобразовать строки в числовые значения с помощью функции `int()` или `float()`. Например:

```
z = int(x) + int(y) # z = 79 (целочисленное сложение)
```

Нестроковые значения можно преобразовать в строковое представление с помощью функции `str()`, `repr()` или `format()`. Например:

```
s = “Значение переменной x: “ + str(x)
s = “Значение переменной x: “ + repr(x)
s = “Значение переменной x: “ + format(x,”4d”)
```

Несмотря на то что обе функции `str()` и `repr()` воспроизводят строки, на самом деле возвращаемые ими результаты обычно немного отличаются. Функция `str()` воспроизводит результат, который дает применение инструкции `print`, тогда как функция `repr()` воспроизводит строку в том виде, в каком она обычно вводится в тексте программы для точного представления значения объекта. Например:

```
>>> x = 3.4
>>> str(x)
‘3.4’
>>> repr(x)
‘3.3999999999999999’
>>>
```

Ошибка округления числа 3.4 в предыдущем примере не является ошибкой Python. Это следствие особенностей представления чисел с плавающей точкой двойной точности, которые не могут быть точно представлены в десятичном формате из-за аппаратных ограничений компьютера.

Для преобразования значений в строку с возможностью форматирования используется функция `format()`. Например:

```
>>> format(x,"0.5f")
'3.40000'
>>>
```

Списки

Списки – это последовательности произвольных объектов. Списки создаются посредством заключения элементов списка в квадратные скобки, как показано ниже:

```
names = [ "Dave", "Mark", "Ann", "Phil" ]
```

Элементы списка индексируются целыми числами, первый элемент списка имеет индекс, равный нулю. Для доступа к отдельным элементам списка используется оператор индексирования:

```
a = names[2]      # Вернет третий элемент списка, "Ann"
names[0] = "Jeff" # Запишет имя "Jeff" в первый элемент списка
```

Для добавления новых элементов в конец списка используется метод `append()`:

```
names.append("Paula")
```

Для вставки элемента в середину списка используется метод `insert()`:

```
names.insert(2, "Thomas")
```

С помощью оператора среза можно извлекать и изменять целые фрагменты списков:

```
b = names[0:2]                      # Вернет ["Jeff", "Mark"]
c = names[2:]                       # Вернет ["Thomas", "Ann", "Phil", "Paula"]
names[1] = 'Jeff'                   # Во второй элемент запишет имя 'Jeff'
names[0:2] = ['Dave','Mark','Jeff'] # Заменит первые два элемента
                                    # списком справа.
```

Оператор сложения (+) выполняет конкатенацию списков:

```
a = [1,2,3] + [4,5] # Создаст список [1,2,3,4,5]
```

Пустой список можно создать одним из двух способов:

```
names = []     # Пустой список
names = list() # Пустой список
```

Список может содержать объекты любого типа, включая другие списки, как показано в следующем примере:

```
a = [1,"Dave",3.14, ["Mark", 7, 9, [100,101]], 10]
```

Доступ к элементам во вложенных списках осуществляется с применением дополнительных операторов индексирования, как показано ниже:

```
a[1]       # Вернет "Dave"
a[3][2]    # Вернет 9
a[3][3][1] # Вернет 101
```

Программа в листинге 1.2 демонстрирует дополнительные особенности списков. Она читает список чисел из файла, имя которого указывается в виде аргумента командной строки, и выводит минимальное и максимальное значения.

Листинг 1.2. Дополнительные особенности списков

```
import sys              # Загружает модуль sys
if len(sys.argv) != 2 : # Проверка количества аргументов командной строки:
    print "Пожалуйста, укажите имя файла"
    raise SystemExit(1)
f = open(sys.argv[1])   # Имя файла, полученное из командной строки
lines = f.readlines()   # Читает все строки из файла в список
f.close()

# Преобразовать все значения из строк в числа с плавающей точкой
fvalues = [float(line) for line in lines]

# Вывести минимальное и максимальное значения
print "Минимальное значение: ", min(fvalues)
print "Максимальное значение: ", max(fvalues)
```

В первой строке этой программы с помощью инструкции `import` загружается модуль `sys` из стандартной библиотеки Python. Этот модуль используется для получения доступа к аргументам командной строки.

Функции `open()` передается имя файла, которое было получено как аргумент командной строки и помещено в список `sys.argv`. Метод `readlines()` читает все строки из файла и возвращает список строк.

Выражение `[float(line) for line in lines]` создает новый список, выполняя обход всех строк и применяя функцию `float()` к каждой из них. Эта конструкция, чрезвычайно удобная для создания списков, называется *генератором списков*. Поскольку строки из файла можно также читать с помощью цикла `for`, программу можно сократить, объединив чтение файла и преобразование значений в одну инструкцию, как показано ниже:

```
fvalues = [float(line) for line in open(sys.argv[1])]
```

После того как входные строки будут преобразованы в список, содержащий числа с плавающей точкой, с помощью встроенных функций `min()` и `max()` отыскиваются минимальное и максимальное значения.

Кортежи

Для создания простейших структур данных можно использовать *кортежи*, которые позволяют упаковывать коллекции значений в единый объект. Кортеж создается заключением группы значений в круглые скобки, например:

```
stock = ('GOOG', 100, 490.10)
```

```
address = ('www.python.org', 80)
person = (first_name, last_name, phone)
```

Интерпретатор Python часто распознает кортежи, даже если они не заключены в круглые скобки:

```
stock = 'GOOG', 100, 490.10
address = 'www.python.org',80
person = first_name, last_name, phone
```

Для полноты можно отметить, что имеется возможность определять кортежи, содержащие 0 или 1 элемент, для чего используется специальный синтаксис:

```
a = ()      # Кортеж с нулевым количеством элементов (пустой кортеж)
b = (item,) # Кортеж с одним элементом (обратите внимание на запятую в конце)
c = item,   # Кортеж с одним элементом (обратите внимание на запятую в конце)
```

Элементы кортежа могут извлекаться с помощью целочисленных индексов, как и в случае со списками. Однако более часто используется прием распаковывания кортежей во множество переменных, как показано ниже:

```
name, shares, price = stock
host, port = address
first_name, last_name, phone = person
```

Кортежи поддерживают практически те же операции, что и списки (такие как доступ к элементам по индексу, извлечение среза и конкатенация), тем не менее содержимое кортежа после его создания невозможно изменить (то есть нельзя изменить, удалить или добавить новый элемент в существующий кортеж). По этой причине кортеж лучше рассматривать как единый объект, состоящий из нескольких частей, а не как коллекцию отдельных объектов, в которую можно вставлять новые или удалять существующие элементы.

Вследствие большого сходства кортежей и списков некоторые программисты склонны полностью игнорировать кортежи и использовать списки, потому что они выглядят более гибкими. Хотя в значительной степени это так и есть, тем не менее, если в программе создается множество небольших списков (когда каждый содержит не более десятка элементов), то они занимают больший объем памяти, по сравнению с кортежами. Это обусловлено тем, что для хранения списков выделяется немного больше памяти, чем требуется, – с целью оптимизировать скорость выполнения операций, реализующих добавление новых элементов. Так как кортежи доступны только для чтения, для их хранения используется меньше памяти, т. к. дополнительное пространство не выделяется.

Кортежи и списки часто используются совместно. Например, следующая программа демонстрирует, как можно организовать чтение данных из файла с переменным количеством столбцов, где значения отделяются друг от друга запятыми:

```
# Файл содержит строки вида "name,shares,price"
filename = "portfolio.csv"
portfolio = []
```

```
for line in open(filename):
    fields = line.split(“,”)  # Преобразует строку в список
    name = fields[0]          # Извлекает и преобразует отдельные значения полей
    shares = int(fields[1])
    price = float(fields[2])
    stock = (name,shares,price) # Создает кортеж (name, shares, price)
    portfolio.append(stock)     # Добавляет в список записей
```

Строковый метод `split()` разбивает строку по указанному символу и создает список значений. Структура данных `portfolio`, созданная этой программой, имеет вид двухмерного массива строк и столбцов. Каждая строка представлена кортежем и может быть извлечена, как показано ниже:

```
>>> portfolio[0]
(‘GOOG’, 100, 490.10)
>>> portfolio[1]
(‘MSFT’, 50, 54.23)
>>>
```

Отдельные значения могут извлекаться следующим способом:

```
>>> portfolio[1][1]
50
>>> portfolio[1][2]
54.23
>>>
```

Ниже приводится простейший способ реализовать обход всех записей и распаковать значения полей в набор переменных:

```
total = 0.0
for name, shares, price in portfolio:
    total += shares * price
```

Множества

Множества используются для хранения неупорядоченных коллекций объектов. Создаются множества с помощью функции `set()`, которой передаются последовательности элементов, как показано ниже:

```
s = set([3,5,9,10]) # Создаст множество чисел
t = set(“Hello”)    # Создаст множество уникальных символов
```

В отличие от кортежей, множества являются неупорядоченными коллекциями и не предусматривают возможность доступа к элементам по числовому индексу. Более того, элементы множества никогда не повторяются. Например, если поближе рассмотреть значения, полученные в предыдущем примере, можно заметить следующее:

```
>>> t
set([‘H’, ‘e’, ‘l’, ‘o’])
```

Обратите внимание, что в множестве присутствует только один символ `‘l’`.

Множества поддерживают стандартные операции над коллекциями, включая объединение, пересечение, разность и симметричную разность. Например:

```
a = t | s # Объединение t и s
b = t & s # Пересечение t и s
c = t - s # Разность (элементы, присутствующие в t, но отсутствующие в s)
d = t ^ s # Симметричная разность (элементы, присутствующие в t или в s,
          # но не в двух множествах сразу)
```

С помощью методов `add()` и `update()` можно добавлять новые элементы в множество:

```
t.add('x')           # Добавит единственный элемент
s.update([10,37,42]) # Добавит несколько элементов в множество s
```

Удалить элемент множества можно с помощью метода `remove()`:

```
t.remove('H')
```

Словари

Словарь – это ассоциативный массив, или таблица хешей, содержащий объекты, индексированные ключами. Чтобы создать словарь, последовательность элементов необходимо заключить в фигурные скобки ({}), как показано ниже:

```
stock = {
    "name" : "GOOG",
    "shares" : 100,
    "price" : 490.10
}
```

Доступ к элементам словаря осуществляется с помощью оператора индексирования по ключу:

```
name = stock["name"]
value = stock["shares"] * stock["price"]
```

Добавление или изменение объектов в словаре выполняется следующим способом:

```
stock["shares"] = 75
stock["date"] = "June 7, 2007"
```

Чаще всего в качестве ключей применяются строки, тем не менее, для этих целей допускается использовать большинство других объектов языка Python, включая числа и кортежи. Определенные объекты, включая списки и словари, не могут использоваться в качестве ключей, потому что их содержимое может изменяться.

Словари обеспечивают удобный способ определения объектов, содержащих именованные поля, как было показано выше. Кроме того, словари могут использоваться, как контейнеры, позволяющие быстро выполнять поиск в неупорядоченных данных. В качестве примера ниже приводится словарь, содержащий цены на акции:

```
prices = {
    "GOOG" : 490.10,
    "AAPL" : 123.50,
    "IBM"  : 91.50,
    "MSFT" : 52.13
}
```

Создать пустой словарь можно одним из двух способов:

```
prices = {}     # Пустой словарь
prices = dict() # Пустой словарь
```

Проверку наличия элемента в словаре можно выполнить с помощью оператора `in`, как показано в следующем примере:

```
if "SCOX" in prices:
    p = prices["SCOX"]
else:
    p = 0.0
```

Данную последовательность действий можно выразить в более компактной форме:

```
p = prices.get("SCOX",0.0)
```

Чтобы получить список ключей словаря, словарь можно преобразовать в список:

```
syms = list(prices) # syms = ["AAPL", "MSFT", "IBM", "GOOG"]
```

Для удаления элементов словаря используется инструкция `del`:

```
del prices["MSFT"]
```

Словари являются, пожалуй, наиболее оптимизированным типом данных в языке Python. Поэтому если в программе необходимо организовать хранение и обработку данных, практически всегда лучше использовать словари, а не пытаться создавать собственные структуры данных.

Итерации и циклы

Для организации циклов наиболее часто используется инструкция `for`, которая позволяет выполнить обход элементов коллекции. Итерации – одна из самых богатых особенностей языка Python. Однако наиболее часто используемой формой итераций является простой цикл по элементам последовательности, такой как строка, список или кортеж. Пример реализации итераций приводится ниже:

```
for n in [1,2,3,4,5,6,7,8,9]:
    print "2 в степени %d = %d" % (n, 2**n)
```

В данном примере на каждой итерации переменная `n` будет последовательно получать значения из списка `[1,2,3,4,...,9]`. Поскольку необходимость организовать цикл по фиксированному диапазону целочисленных значений возникает достаточно часто, для этих целей используется сокращенная форма записи:

```
for n in range(1,10):
    print "2 в степени %d = %d" % (n, 2**n)
```

Функция `range(`*i*`,`*j* `[,`*stride*`])` создает объект, представляющий диапазон целых чисел со значениями от *i* по *j*-1. Если начальное значение не указано, оно берется равным нулю. В третьем необязательном аргументе *stride* можно передать шаг изменения значений. Например:

```
a = range(5)      # a = 0,1,2,3,4
b = range(1,8)    # b = 1,2,3,4,5,6,7
c = range(0,14,3) # c = 0,3,6,9,12
d = range(8,1,-1) # d = 8,7,6,5,4,3,2
```

Будьте внимательны при использовании функции `range()` в Python 2, так как в этой версии интерпретатора она создает полный список значений. Для очень больших диапазонов это может привести к ошибке нехватки памяти. Поэтому при работе с ранними версиями Python программисты используют альтернативную функцию `xrange()`. Например:

```
for i in xrange(100000000): # i = 0,1,2,...,99999999
    инструкции
```

Функция `xrange()` создает объект, который вычисляет очередное значение только в момент обращения к нему. Именно поэтому данный способ является более предпочтительным при работе с большими диапазонами целых чисел. В версии Python 3 функция `xrange()` была переименована в `range()`, а прежняя реализация функции `range()` была удалена.

Возможности инструкции `for` не ограничиваются последовательностями целых чисел, она также может использоваться для реализации итераций через объекты самых разных типов, включая строки, списки, словари и файлы. Например:

```
a = "Привет, Мир"
# Вывести отдельные символы в строке a
for c in a:
    print c

b = ["Dave","Mark","Ann","Phil"]
# Вывести элементы списка
for name in b:
    print name

c = { 'GOOG' : 490.10, 'IBM' : 91.50, 'AAPL' : 123.15 }
# Вывести элементы словаря
for key in c:
    print key, c[key]

# Вывести все строки из файла
f = open("foo.txt")
for line in f:
    print line,
```

Цикл `for` является одной из самых мощных особенностей языка Python, так как позволяет вам создавать собственные объекты-итераторы и функции-генераторы, которые возвращают последовательности значений. Подроб-

нее об итераторах и генераторах рассказывается в следующей главе, а также в главе 6 «Функции и функциональное программирование».

Функции

Создание функции производится с помощью инструкции `def`, как показано в следующем примере:

```
def remainder(a, b):
    q = a // b      # // - оператор деления с усечением дробной части.
    r = a - q*b
    return r
```

Чтобы вызвать функцию, достаточно указать ее имя и следующий за ним список аргументов, заключенный в круглые скобки, например: `result = remainder(37, 15)`. Если потребуется вернуть из функции несколько значений, можно использовать кортеж, как показано ниже:

```
def divide(a, b):
    q = a // b    # Если a и b - целые числа, q будет целым числом
    r = a - q*b
    return (q,r)
```

Когда функция возвращает кортеж с несколькими значениями, его легко можно распаковать в множество отдельных переменных, как показано ниже:

```
quotient, remainder = divide(1456, 33)
```

Присвоить аргументу функции значение по умолчанию можно с помощью оператора присваивания:

```
def connect(hostname, port, timeout=300):
    # Тело функции
```

Если в определении функции для каких-либо параметров указаны значения по умолчанию, при последующих вызовах функции эти параметры можно опустить. Если при вызове какой-то из этих параметров не указан, он получает значение по умолчанию. Например:

```
connect('www.python.org', 80)
```

Также имеется возможность передавать функции именованные аргументы, которые при этом можно перечислять в произвольном порядке. Однако в этом случае вы должны знать, какие имена аргументов указаны в определении функции. Например:

```
connect(port=80, hostname="www.python.org")
```

Когда внутри функции создаются новые переменные, они имеют локальную область видимости. То есть такие переменные определены только в пределах тела функции, и они уничтожаются, когда функция возвращает управление вызывающей программе. Чтобы иметь возможность изменять глобальные переменные внутри функции, эти переменные следует определить в теле функции с помощью инструкции `global`:

```
count = 0
...
def foo():
    global count
    count += 1     # Изменяет значение глобальной переменной count
```

Генераторы

Вместо единственного значения функция, с помощью инструкции `yield`, может генерировать целые последовательности результатов. Например:

```
def countdown(n):
    print "Обратный отсчет!"
    while n > 0:
        yield n   # Генерирует значение (n)
        n -= 1
```

Любая функция, которая использует инструкцию `yield`, называется *генератором*. При вызове функции-генератора создается объект, который позволяет получить последовательность результатов вызовом метода `next()` (или `__next__()` в Python 3). Например:

```
>>> c = countdown(5)
>>> c.next()
Обратный отсчет!
5
>>> c.next()
4
>>> c.next()
3
>>>
```

Метод `next()` заставляет функцию-генератор выполняться, пока не будет достигнута следующая инструкция `yield`. После этого метод `next()` возвращает значение, переданное инструкции `yield`, и выполнение функции приостанавливается. При следующем вызове метода `next()` функция продолжит выполнение, начиная с инструкции, следующей непосредственно за инструкцией `yield`. Этот процесс продолжается, пока функция не вернет управление.

Как правило, метод `next()` не вызывается вручную, как это было показано выше. Вместо этого функция-генератор обычно используется в инструкции цикла `for`, например:

```
>>> for i in countdown(5):
...     print i,
Обратный отсчет!
5 4 3 2 1
>>>
```

Генераторы обеспечивают чрезвычайно широкие возможности при конвейерной обработке или при работе с потоками данных. Например, следующая функция-генератор имитирует поведение команды `tail -f`, которая ча-

сто используется в операционной системе UNIX для мониторинга файлов журналов:

```
# следит за содержимым файла (на манер команды tail -f)
import time
def tail(f):
    f.seek(0,2)             # Переход в конец файла
    while True:
        line = f.readline() # Попытаться прочитать новую строку текста
        if not line:        # Если ничего не прочитано,
            time.sleep(0.1) # приостановиться на короткое время
            continue        # и повторить попытку
        yield line
```

Ниже приводится пример генератора, который отыскивает определенную подстроку в последовательности строк:

```
def grep(lines, searchtext):
    for line in lines:
        if searchtext in line: yield line
```

Ниже приводится пример, в котором объединены оба эти генератора с целью реализовать простейшую конвейерную обработку:

```
# Реализация последовательности команд "tail -f | grep python"
# на языке Python
wwwlog  = tail(open("access-log"))
pylines = grep(wwwlog,"python")
for line in pylines:
    print line,
```

Одна из важных особенностей генераторов состоит в том, что они могут использоваться вместо других итерируемых объектов, таких как списки или файлы. В частности, когда вы пишете такую инструкцию, как `for` *`item`* `in` *`s`*, имя *`s`* может представлять список элементов, строки в файле, результат вызова функции-генератора или любой другой объект, поддерживающий итерации. Возможность использования самых разных объектов под именем *`s`* может оказаться весьма мощным инструментом создания расширяемых программ.

Сопрограммы

Обычно функции оперируют единственным набором входных аргументов. Однако функцию можно написать так, что она будет действовать, как программа, обрабатывающая последовательность входных данных. Такие функции называются *сопрограммами*, а создаются они с помощью инструкции `yield`, используемой в выражении `(yield)`, как показано в следующем примере:

```
def print_matches(matchtext):
    print "Поиск подстроки", matchtext
    while True:
        line = (yield)        # Получение текстовой строки
```